Hans Grönniger ; Holger Krahn ; Bernhard Rumpe ; Martin Schindler ; Steven Völkel

MontiCore 1.0 : Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen

(Informatik-Bericht ; 2006-04)

Technische Universität Braunschweig
Institut für Software Systems Engineering
Informatik-Bericht 2006-04

MontiCore 1.0

Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen

Hans Grönniger
Holger Krahn
Bernhard Rumpe
Martin Schindler
Steven Völkel

Braunschweig, den 23. August 2006

Kurzfassung

Dieser technische Bericht beschreibt das Modellierungsframework MontiCore in Version 1.0, das zur Erstellung und Verarbeitung von domänenspezifischen Sprachen dient. Dabei werden domänenspezifische Sprachen zur Softwareentwicklung und zur Informationsverarbeitung genutzt. In diesem Handbuch wird zunächst eine kurze Einführung in eine agile Softwareentwicklungsmethodik gegeben, die auf domänenspezifischen Sprachen basiert, dann werden technische Grundlagen einer solchen Entwicklung in MontiCore erklärt. Zu diesem Zweck wird zunächst ein einfaches Beispiel beschrieben anhand dessen der Funktionsumfang der MontiCore-Grammatikbeschreibungssprache und der MontiCore-Klassenbeschreibungssprache erläutert wird. Ergänzend werden komplexere Beispiele kurz aufgezeigt und das entwickelte Framework mit anderen Ansätzen verglichen.
MontiCore bietet in der derzeitigen Fassung vor allem Unterstützung zur schnellen Definition textbasierter domänenspezifischer Sprachen. Dazu gehören die Definition der kontextfreien Sprache, die Generierung eines Parsers, einer Datenstruktur zur Speicherung des abstrakten Syntaxbaums und zur Bearbeitung der abstrakten Syntax. Zusätzliche sogennante „Konzepte“ erlauben die Behandlung von Symboltabellen, Kontextbedingungen, Namespaces etc. MontiCore bietet in der aktuellen Fassung insbesondere bereits (zumindest eingeschränkt) Unterstützung zur Komposition von Sprachteilen, so dass die Entwicklung neuer domänenspezifischer Sprachen durch Wiederverwendung deutlich effizienter wird. Aus diesem Grund sind auch bereits mehrere Sprachen definiert worden, von denen einige hier als Beispiele vorgestellt werden. MontiCore ist selbst teilweise im Rahmen eines Bootstrapping-Prozesses mittels solcher Sprachen definiert worden.

Inhaltsverzeichnis

1. Einleitung

Die Informatik kennt derzeit zwei große Grundströmungen zur Verbesserung von Qualität und Effizienz in der Softwareentwicklung. Zum einen konzentrieren sich agile Methoden vor allem darauf, essentielle Aktivitäten der Softwareentwicklung zu intensivieren und andere, weniger wichtige, Aktivitäten zu reduzieren. Zum anderen zeigt sich, dass die durchgängige Verwendung von Modellen beginnend mit der Anforderungserhebung bis hin zur Implementierung einen wesentlichen Vorteil bei Qualität und meist auch Effizienz bringt. Im Kontext der Geschäftssysteme hat sich die UML [WWWz] als defacto-Standard weitgehend durchgesetzt. Im eingebetteten Bereich konkurrieren derzeit sehr erfolgreich Ansätze auf Basis von Matlab/Simulink/Stateflow [ABRW05], SystemC [BD04] mit Realtime-Varianten der UML (z.B. [Dou04]) und einem Derivat der UML mit dem Namen SysML [WWWy]. Neben diesen Sprachstandards gibt es eine Reihe applikationsspezifischer und technischer Sprachen, die für bestimmte Zwecke höchst effektiv eingesetzt werden können. Dazu zählen etwa Konfigurationssprachen für ERP-Systeme [DRvdA+05], Definitionssprachen für Tests [BEK97, Gra00], Sprachen für die Abwicklung von Kompilation und Deployment (Make [Her03] oder Ant [WWWa]), Compiler-Compiler [LMB92, DS03, PQ95] und dergleichen mehr.

Trotz der Standardisierung der UML ist gerade durch die Möglichkeit der Profilbildung für die UML absehbar, dass es eine größere Zahl von Derivaten der UML für spezifische applikationsorientierte und technische Domänen geben wird [FPR02, Coo00]. Damit verschwimmt die scharfe Grenze zu den domänenspezifischen Sprachen (DSL [Cza05]), denen häufig ähnliche Paradigmen zugrunde liegen wie den Teilsprachen der UML, die aber unabhängig entstanden und derzeit meist besser auf spezifische Problemstellungen zugeschnitten sind. Weil auch der Wunsch nach einer schnellen und effektiven Anpassung von vorhandenen domänenspezifischen Sprachen oder die Neuentwicklung einer domänenspezifischen Sprache für eine neue Applikationsdomäne oder Problemstellung immer häufiger nachgefragt wird, wurde das Framework MontiCore in seiner aktuellen Fassung 1.0 entwickelt.

Aufgabe von MontiCore ist es, die Nutzung von Modellen ins Zentrum der Softwareentwicklung zu rücken und davon ausgehend ebenfalls für die Verarbeitung komplexer Informationen verfügbar zu machen. Modellbasierte Softwareentwicklung [Rum04b, Rum04a] nutzt, wie Abbildung 1.1 zeigt, Modelle nicht nur zur Dokumentation und Kommunikation zwischen den Entwicklern, sondern zur Generierung von Code und zur automatisierten Entwicklung von Tests, gegebenenfalls um mit Rapid

Prototyping agiler und effizienter zu einem lauffähigen System zu gelangen, durch statische Analysen auf Modellebene die Modell- und die Codequalität früher und effektiver sicherstellen zu können und letztendlich bereits auf Modellebene die Weiterentwicklung von Software zu planen und umzusetzen (Refactoring).

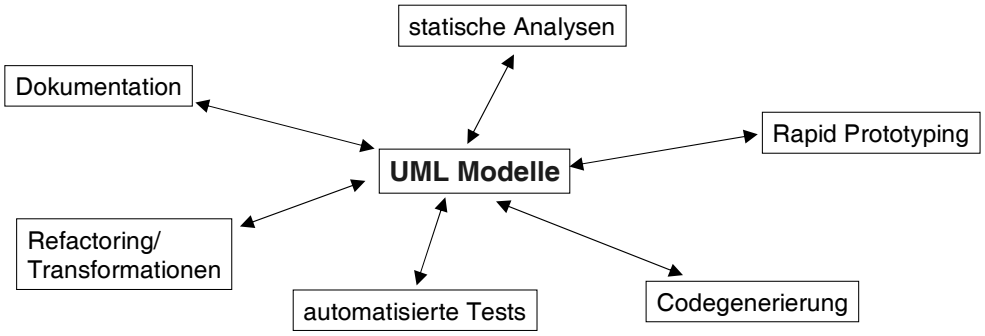


Abbildung 1.1.: Modellbasierte Softwareentwicklung

Die mit dem MontiCore-Framework verfügbaren Werkzeuge sind primär dafür gedacht, Modellierungssprachen schnell und effektiv zu beschreiben, Werkzeugunterstützung zur Verarbeitung zu definieren sowie Analyse- und Generierungswerkzeuge dafür „of the shelf“ zur Verfügung zu stellen. Ein Standardanwendungsfall sind dabei relevante Teile der UML, die mit ihren derzeit 13 Notationsarten eine reichhaltige Sammlung an Konzepten mit konkreter syntaktischer Ausprägung zur Verfügung stellt. Manche domänenspezifische Sprachen können durch Anpassung vorhandener UML-Teilnotationen definiert werden, andere sind grundlegend neu festzulegen. MontiCore stellt für beide Einsatzszenarien eine entsprechende Infrastruktur zur Verfügung.

Für einen effektiven Einsatz des MontiCore-Frameworks in der Softwareentwicklung ist es essentiell, die verwendete Methodik entsprechend anzupassen. Der Einsatz eines auf MontiCore basierenden Code- oder Testfallgenerators impliziert auf jeden Fall eine entsprechende Methodik. Im Idealfall besteht die Softwareentwicklung in Zukunft nur noch geringfügig aus der klassichen Entwicklung technischer Komponenten, z.B. auf Java- oder C++-Basis, und basiert primär auf der Wiederverwendung und Weiterentwicklung einer geeigneten DSL, der Modellierung von höherwertigen Funktionen oder Aspekten unter Nutzung dieser DSL und der Generierung von Code aus der DSL, der die verfügbaren technischen Komponenten instanziiert und verschaltet. Solche DSLs stellen dann auch die Basis für frühe Modellbildungen, Analysen und letztlich der Qualitätssicherung dar. Neben der Wiederverwendung von domänenspezifischen MontiCore-Anwendungen dürfte in größeren Projekten normalerweise eine Anpassung der Werkzeuglandschaft sinnvoll sein. Um dieser Anpassung Rechnung zu tragen, ist das MontiCore-Werkzeugset in besonders schlanker und offener Weise als Framework realisiert.

Entsprechend sind die primären Entwurfsentscheidungen des MontiCore-Toolsets:

- Die MontiCore-Engine ist in mehreren Java-Teilkomponenten realisiert, die in offener Weise erweitert und ergänzt werden können.
- Die Eingabe für die MontiCore-Engine ist primär textbasiert, d.h. Modelle werden in textueller Form erstellt, gespeichert und dem MontiCore-System zugänglich gemacht. Ein solches Vorgehen erlaubt eine deutlich schnellere Erstellung und Weiterentwicklung einer DSL nebst zugehörigem Editor als es ein graphikbasiertes System heute erlauben würde. Darüber hinaus können mit Versionskontrolle und einer nicht interaktiven Nutzung des MontiCore-Systems moderne Softwareengineering-Prinzipien wie etwa das Daily Build, Versions- und Konfigurationsmanagement in gewohnter Weise eingesetzt werden. Mit Eclipse [DFK+04] kann relativ einfach ein syntaxgesteuerter Editor erstellt werden. Die Nutzung vorhandener Editoren, wie etwa UML-Werkzeugen oder eigener Entwicklungen als graphisches Frontend ist darüber hinaus ebenfalls möglich.
- Die Eingabesprache der MontiCore-Engine kann kompositional zusammengestellt werden. So ist es möglich, Zustandsmaschinen mit Java, OCL oder einer beliebigen (vorhandenen) anderen Sprache für Bedingungen und Anweisungen zu kombinieren und so Wiederverwendung von Sprachteilen zu betreiben.
- Die MontiCore-Komponenten sind ebenfalls kompositional in dem Sinne, dass einzelne Analyse- und Generierungsalgorithmen unter Nutzung publizierter Schnittstellen einfach zusammengestellt werden können.
- MontiCore nutzt Elemente klassischen Compilerbaus [ASU86, WWWm] und der Model Driven Architecture (MDA) [WWWx, MSU04], um bewährte Technologien mit innovativen Konzepten zu vereinen. MontiCore ist partiell im Bootstrapping-Verfahren entwickelt worden. So steht eine DSL zur Definition von Grammatiken zur Verfügung, die gleichzeitig einen abstrakten Syntaxbaum und eine DSL zur Generierung von Klassenstrukturen aufbaut.

Die in MontiCore eingesetzen DSLs werden in diesem Bericht jeweils als eigenständige Sprachen beschrieben und dienen so gleichzeitig zum Einsatz in MontiCore und als DSL-Sprachbeispiele. MontiCore nutzt die unterstützten Sprachen selbst und ist deshalb in einem Bootstrapping-Verfahren realisiert.

Darüber hinaus ist geplant oder in Arbeit, eine Reihe weiterer Sprachen wie etwa Zustandsmaschinen, die wesentlichsten Teilsprachen der UML, Petrinetze in einfachen Versionen, sowie eine ausgereifte Architekturbeschreibungssprache (verwandt mit UML-Kompositions-Struktur-Diagrammen) zur Verfügung zu stellen, auf deren Basis eigene Sprachdefinitionen erfolgen oder in die eigene Sprachen transformiert werden können.

Dieser technische Bericht gliedert sich wie folgt: Kapitel 2 beinhaltet eine Tutorial-ähnliche Einführung in die Benutzung von MontiCore. Darin wird anhand eines einfachen Beispieles die Definition einer neuen DSL erklärt. Darauf aufbauend wird eine einfache Methode zur Transformation von DSLs dargestellt.

In Kapitel 3 wird eine Übersicht über grundlegende Mechanismen der generativen Entwicklung von Software mit MontiCore dargestellt. Aufbauend auf dieser Grundlage werden in den Kapiteln 4 und 5 die Kernkomponenten von MontiCore, die MontiCore-Klassenbeschreibungssprache und das MontiCore-Grammatikformat genauer vorgestellt. Sie sind selbst als DSLs mit Monticore definiert und können daher ebenfalls als Beispiele verstanden werden.

In Kapitel 6 werden weitere DSL-Fallbeispiele vorgestellt, um die Nutzung des MontiCore-Frameworks zu demonstrieren. Kapitel 7 gibt einen Überblick über verwandte Ansätze und Kapitel 8 schließt die Arbeit ab.

2. Nutzung von MontiCore

Das Werkzeugframework MontiCore dient dem schnellen und effektiven Erstellen von domänenspezifischen Sprachen (DSL) in der Softwareentwicklung. Hierzu verarbeitet MontiCore eine erweiterte Grammatikbeschreibung der DSL und generiert daraus Komponenten zur Verarbeitung von in der DSL verfassten Dokumenten. Erzeugt werden zum Beispiel Parser, AST-Klassen und einfache Symboltabellen. Auf diese Weise können schnell eigene Sprachen definiert und zusammen mit MontiCore als DSL-spezifisches Werkzeug genutzt werden. In diesem Kapitel wird vorgestellt, wie eine DSL mit Hilfe des MontiCore-Frameworks entwickelt wird. Da dieses Dokument nur eine Übersicht liefert, werden die verwendeten MontiCore-Komponenten nur knapp beschrieben. Details sind jeweils der Dokumentation dieser Komponenten zu entnehmen.

Um den Einstieg in MontiCore zu erleichtern, werden die erforderlichen Schritte anhand eines kleinen Beispiels demonstriert. Die zu entwickelnde DSL erlaubt XY-Koordinaten zu modellieren. Im Beispiel wird der Entwurf der Grammatik, der Umgang mit MontiCore und die Implementierung einer Modelltransformation gezeigt.

2.1. Installation von MontiCore

Die für die Verwendung von MontiCore notwendigen Dateien sind auf der MontiCore-Homepage [WWWu] zu finden. Dabei werden zwei unterschiedliche Arten der Nutzung unterschieden:

Nutzung als Standalone-Werkzeug
: Der Kern von MontiCore besteht aus einer jar-Datei (zurzeit de.monticore_-1.0.0.jar für die Entwicklungsumgebung und de.monticore.re_1.0.0.jar für die Laufzeitumgebung), die eine Nutzung über die Kommandozeile ermöglichen - ein installiertes Java SDK 5.0 und Apache Ant vorausgesetzt (siehe Abschnitt 2.1.1). Die Kommandozeilenstruktur ist dazu gedacht, in komplexere automatisierte Werkzeugketten eingebunden zu werden, um damit einen agilen Softwareentwicklungsprozess optimal zu unterstützen.

Nutzung in Eclipse
: MontiCore kann als Eclipse-Plugin verwendet werden. Die Funktionalität entspricht dabei der direkten Nutzung über die Kommandozeile. Jedoch wird mit

Hilfe von Eclipse eine komfortablere Infrastruktur zur Verfügung gestellt und so die Entwicklung von domänenspezifischen Sprachen mit MontiCore erleichtert.

2.1.1. Installation als Standalone-Werkzeug

Wird MontiCore als Standalone-Werkzeug genutzt, ist die Installation folgender Softwareprodukte Voraussetzung:

- Apache Ant ab 1.6 [WWWa]
- Java SDK 5.0 [WWWq]
- GraphViz 2.8 [WWWn]

Zusätzlich muss das `bin`-Verzeichnis von Ant und GraphViz sowie das `bin`- und `lib`-Verzeichnis des Java SDKs in den Pfad (d.h. in der PATH-Variablen) des Betriebssystems aufgenommen werden.

Für die Anwendung von MontiCore müssen die Dateien de.monticore_1.0.0.jar und de.monticore.re_1.0.0.jar in den CLASSPATH eingebunden oder explizit beim Aufruf angegeben werden. Für die Nutzung der Beispiele und die Projekterzeugung im Allgemeinen muss außerdem noch die de.monticore.examples_1.0.0.jar mit aufgenommen werden.

Die Verwendung von MontiCore als Standalone-Werkzeug wird in Abschnitt 2.3 beschrieben.

2.1.2. Installation des Eclipse-Plugins

Für die Nutzung des Eclipse-Plugins müssen folgende Softwareprodukte installiert sein:

- Eclipse 3.1 oder höher [WWWd]
- Java SDK 5.0 [WWWq]
- GraphViz 2.8 [WWWn]

Im Vergleich zur Installation als Standalone-Werkzeug (siehe Abschnitt 2.1.1) muss hier nur das `bin`-Verzeichnis von GraphViz in den Pfad des Betriebssystems aufgenommen werden. Des Weiteren werden einige Einstellungen in Eclipse empfohlen, die über „Window > Preferences“ zugänglich sind:

- unter „General > Workspace“ „Refresh automatically“ anwählen
- unter „Java > Compiler“ „Compiler compliance level“ auf 5.0 stellen (zwingend notwendig)
- unter „Java > Build Path“ „Folders“ anwählen
- unter „Ant > Runtime > Classpath > Global Entries“ `tools.jar` aus dem lib-Verzeichnis des Java SDK und `org.junit_3.8.1/junit.jar` aus dem Plugin-Verzeichnis von Eclipse hinzufügen

Das MontiCore-Eclipse-Plugin besteht aus vier jar-Dateien (zurzeit de.monticore-_1.0.0, de.monticore.re_1.0.0, de.monticore.plugin_1.0.0 und de.monticore.examples_1.0.0), wobei letztere optional ist und nur eingebunden werden muss, wenn Beispielprojekte zur Anwendung von MontiCore gewünscht sind (eine kurze Beschreibung der Beispielprojekte findet sich in Kapitel 6). Um das Plugin zu installieren, müssen alle Instanzen von Eclipse geschlossen und anschließend die vier jar-Dateien in das „plugins“-Verzeichnis der Eclipse-Installation kopiert werden. Beim nächsten Start von Eclipse ist MontiCore installiert und kann wie im Abschnitt 2.2 beschrieben verwendet werden.

2.2. Nutzung in Eclipse

2.2.1. Eröffnung eines MontiCore-Projekts

Wir erstellen ein neues MontiCore-Projekt über folgende Schritte:

Schritt 1
Über „File > New > Project > MontiCore > MontiCore-Project“ öffnen wir den Projekt-Wizard von MontiCore (siehe Abbildung 2.1) und wählen „Next“.

Schritt 2
Nun müssen wir den Projektnamen für die geplante DSL vergeben. Für das hier vorgestellte Beispiel wählen wir `coordDSL` (siehe Abbildung 2.2).

Schritt 3
Nach der Auswahl von „Finish“ erscheint ein neues Projekt `coordDSL` im Package Explorer von Eclipse (Abbildung 2.3).

Wird anstatt „Finish“ in Abbildung 2.2 ein weiteres Mal „Next“ gewählt, erscheint eine Reihe von Beispielen für den Einstieg in MontiCore, die in das neue Projekt integriert werden können (siehe auch Kapitel 6). Wie Abbildung 2.4 zeigt, ist hier

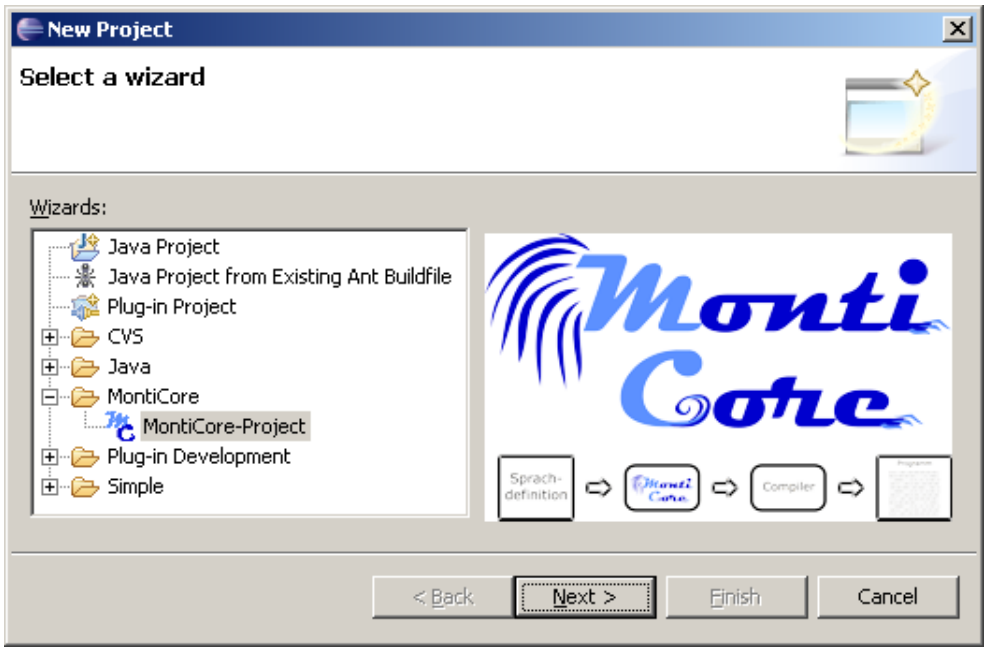


Abbildung 2.1.: Erstellung eines neuen MontiCore-Projektes - Schritt 1

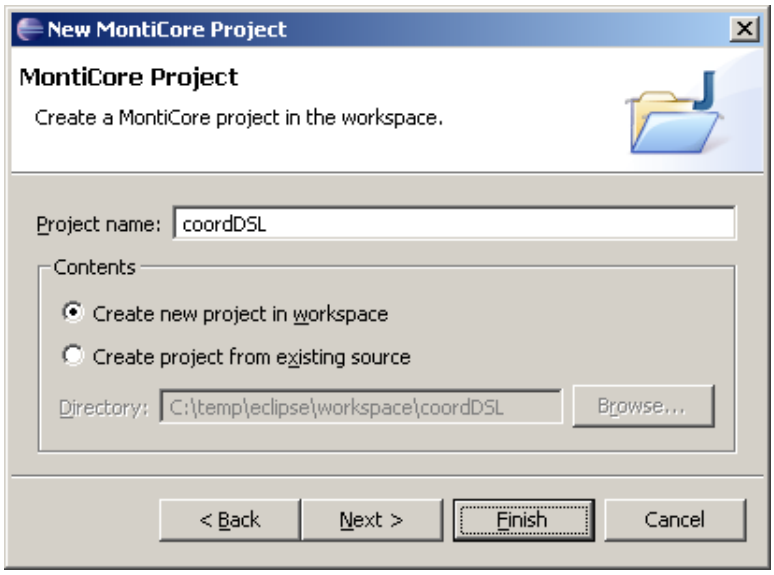


Abbildung 2.2.: Erstellung eines neuen MontiCore-Projektes - Schritt 2

unter anderem auch das in diesem Kapitel vorgestellte Beispiel der coord-DSL vorhanden. Da wir uns aber ansehen wollen, wie eine DSL mit MontiCore Schritt für Schritt erstellt wird, lassen wir diese Möglichkeit außer Acht und erstellen ein leeres MontiCore-Projekt. Für weitere Experimente mit MontiCore können diese Beispiele

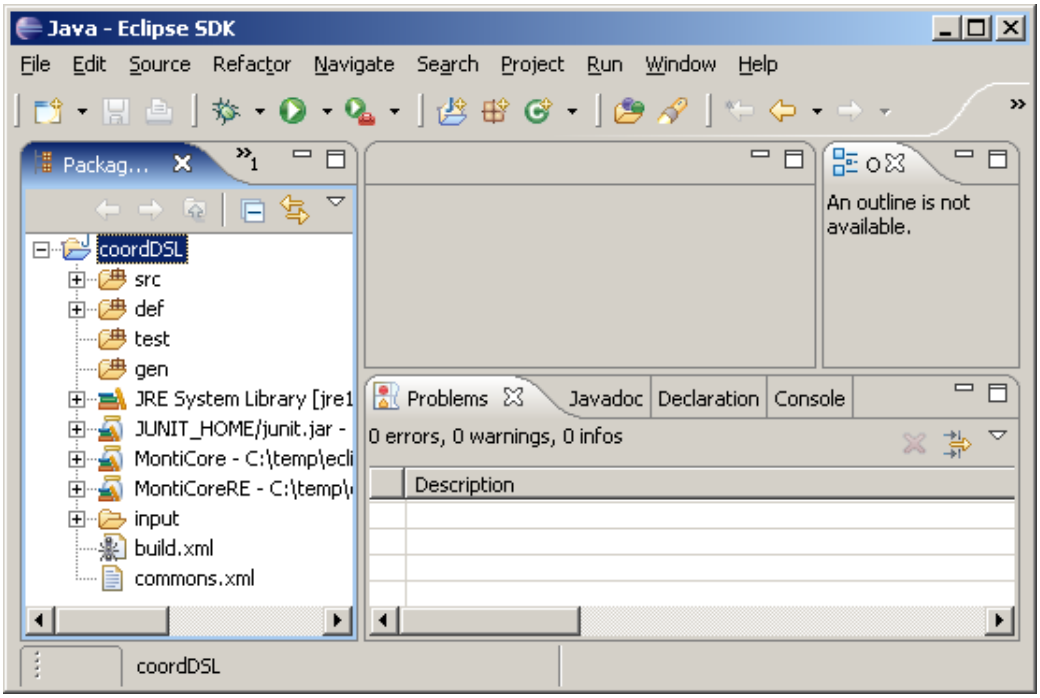


Abbildung 2.3.: Erstellung eines neuen MontiCore-Projektes - Ergebnis

aber einen geeigneten Ausgangspunkt bilden.

In Abbildung 2.3 ist zu sehen, dass unser neu erstelltes MontiCore-Projekt auch ohne die Auswahl eines Beispiels schon eine vorgegebene Ordnerstruktur enthält. Zu dieser legen wir über „File > New > Folder“ zusätzlich einen Ordner `output` für die Ausgaben unserer DSL an. Die vom Projekt-Wizard erzeugte Ordnerstruktur hat folgende Bedeutung:

src - für handcodierten Quellcode
: Wenn wir eine DSL entwerfen, wollen wir nicht nur die Sprache definieren, sondern im Allgemeinen auch Funktionalität mit dieser Sprache verbinden können. Hierzu zählt etwa die Übersetzung in eine Programmiersprache wie Java, um eine ausführbare DSL zu erhalten, oder die Entwicklung von Transformationen (siehe Abschnitte 2.2.6 und 2.2.7). Hierfür ist die Entwicklung von handcodiertem Java-Quellcode notwendig, der in diesem Ordner abgelegt werden soll.

test - für Testfälle
: Um die Qualität des handcodierten Quellcodes in `src` sicherzustellen, sollten wir für diesen Code Testfälle schreiben und im Ordner `test` zusammenfassen. So können wir bei Änderungen leicht überprüfen, ob die getestete Funktionalität erhalten geblieben ist.

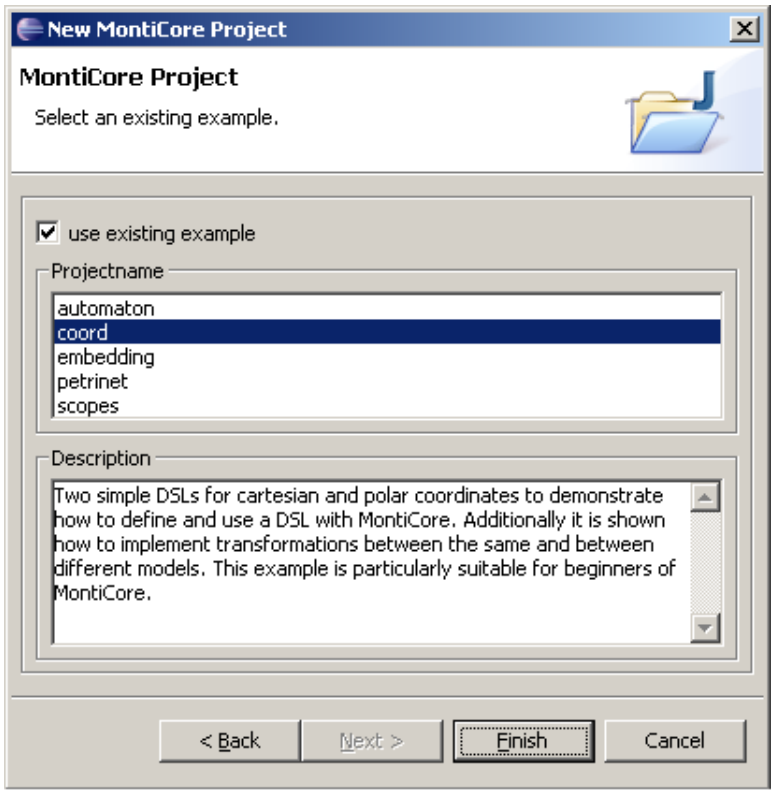


Abbildung 2.4.: Auswahl von Beispielen bei der MontiCore-Projekt-Erstellung

gen - für generierten Quellcode
Dieser Ordner ist dafür gedacht, generierten Code aufzunehmen und so generierten und manuell geschriebenen Code deutlich zu trennen. Hier wird die durch MontiCore generierte Infrastruktur abgelegt (siehe Abschnitt 2.2.4). Sie dient zum Parsen und Verarbeiten von Dokumenten der DSL.

def - für die DSL definierenden Quellcode
Hier wird die Grammtik abgelegt, die unsere DSL definiert (siehe Abschnitt 2.2.3).

input - für in der DSL geschriebenen Quellcode
Hier können wir Quellcode ablegen, der in der DSL selbst geschrieben ist. Die hier abgelegten Modelle werden durch das generierte Werkzeug verarbeitet.

Für die Generierung der Infrastruktur für die zu entwerfende DSL werden außerdem drei Dateien genutzt, die bereits bei der Projekterzeugung angelegt wurden (siehe Abbildung 2.3). `build.xml` ist eine Ant-Datei, deren default-target `compile` unter Eclipse über einen Rechtsklick auf die Datei mit Auswahl von „Run As > Ant Build“ ausgeführt werden kann. Nicht-projektspezifische Targets sind dabei in der

`commons.xml` ausgelagert, die in der `build.xml` importiert wird. Eine weitere Möglichkeit der Generierung bietet die Ausführung der Datei `Generate.java` im Default Package von `src`. Diese kann auch außerhalb von Eclipse für die Generierung eingesetzt werden (siehe dazu Abschnitt 2.3). Die Ant-Datei bietet darüber hinaus noch zusätzliche Möglichkeiten über die folgenden Targets:

- `clean`: Löschen aller generierten Dateien
- `compile`: default-target für die Generierung und Compilierung der Infrastruktur für die Nutzung der DSL
- `generate`: Erzeugung der generierten Dateien ohne anschließender Compilierung
- `plugin`: Erzeugung eines Eclipse-Plugins aus der DSL

Weitere Informationen zur Verwendung der `build.xml` findet sich im Abschnitt 2.2.4.

Mit Ausnahme des Ordners `src`, der die Datei `Generate.java` enthält, sind alle automatisch erstellten Unterordner direkt nach der Projekterzeugung leer. Da somit auch noch keine Grammatikbeschreibung einer DSL existiert, haben die oben beschriebenen Aufrufe noch keine Auswirkung. Unser nächstes Ziel ist demnach die Erstellung einer solchen Grammatik für die Koordinaten-DSL.

2.2.2. Entwurf der DSL anhand eines Beispiels

Bisherige Erfahrungen zeigen, dass sich die Grammatik einer DSL relativ einfach aus konkreten Beispielen ableiten lässt. Abbildung 2.5 zeigt eine mögliche Eingabe unserer DSL für kartesische Koordinaten.

```
1 (2,4) (5,2) (1,7)
```

Abbildung 2.5.: coordinates.cart - Beispieleingabe für die CoordCartesian-DSL

Um später die Grammatik und den daraus generierten Parser zu testen, legen wir die Beispieleingabe aus Abbildung 2.5 als `coordDSL/input/coordinates.cart` ab. In Eclipse erzeugen wir eine neue Textdatei, indem wir über „File > New > File“ den entsprechenden Namen vergeben und den Ordner für die neue Datei auswählen.

2.2.3. Entwurf der Grammatik

Anhand des Beispiels aus Abbildung 2.5 können wir die Grammatikregeln für die Sprache ableiten. Die Formulierung der Grammatik für die Koordinaten-DSL erfolgt in der MontiCore-Grammatikbeschreibungssprache (siehe Kapitel 5) und ist in Abbildung 2.6 zu sehen.

```
1  package mc.coord.cartesian;
2
3  grammar Coordcartesian {
4
5      options {
6          parser lookahead=3
7          lexer lookahead=2
8      }
9
10     ident INT "('0'..'9')+";
11
12     CoordinateFile = (Coordinates:Coordinate)+;
13
14     Coordinate     = "(" X:INT "," Y:INT ")";
15 }
```

Abbildung 2.6.: cartesian.mc - Grammatik der CoordCartesian DSL

Die Zeilen der Grammatik in Abbildung 2.6 haben die folgende Bedeutung:

1. Das Package, zu dem die Grammatik gehört. Unter diesem werden auch die AST-Klassen und der Parser abgelegt.

3. Der Name der Grammatik, der auch den Namen des generierten Parsers bestimmt.

5. Beginn eines Blocks von Optionen, über die das Verhalten von Parser und Lexer beeinflusst werden kann.

6. Angabe des Look-Aheads für den Parser.

7. Angabe des Look-Aheads für den Lexer.

10. Definition des einzigen hier benutzten Terminalsymbols mit dem Namen `INT`. Dabei wird ein regulärer Ausdruck verwendet, dessen Syntax von Antlr bestimmt wird. Dieses Terminalsymbol besteht aus einer oder mehreren Ziffern, die eine natürliche Zahl bilden.

12. Die Regel mit dem Nichtterminal-Symbol `CoordinateFile` beschreibt, dass eine Koordinatendatei aus einer oder mehreren Koordinaten besteht, wobei `Coordinates` einem zum Zugriff im AST bestimmten Attributnamen und `Coordinate` einem Nichtterminal-Symbol entspricht.

14. Die Regel für das Nichtterminal `Coordinate` beschreibt das Aussehen einer kartesischen Koordinate.

Um diese Grammatikbeschreibung zu speichern, erzeugen wir zuerst ein neues Package über „File > New > Package“, wobei wir als Source Folder `coordDSL/def` und als Namen `mc.coord.cartesian` wählen. Mit „File > New > File“ speichern wir in diesem Package die Grammatik als neue Datei `cartesian.mc` (siehe Abbildung 2.7).

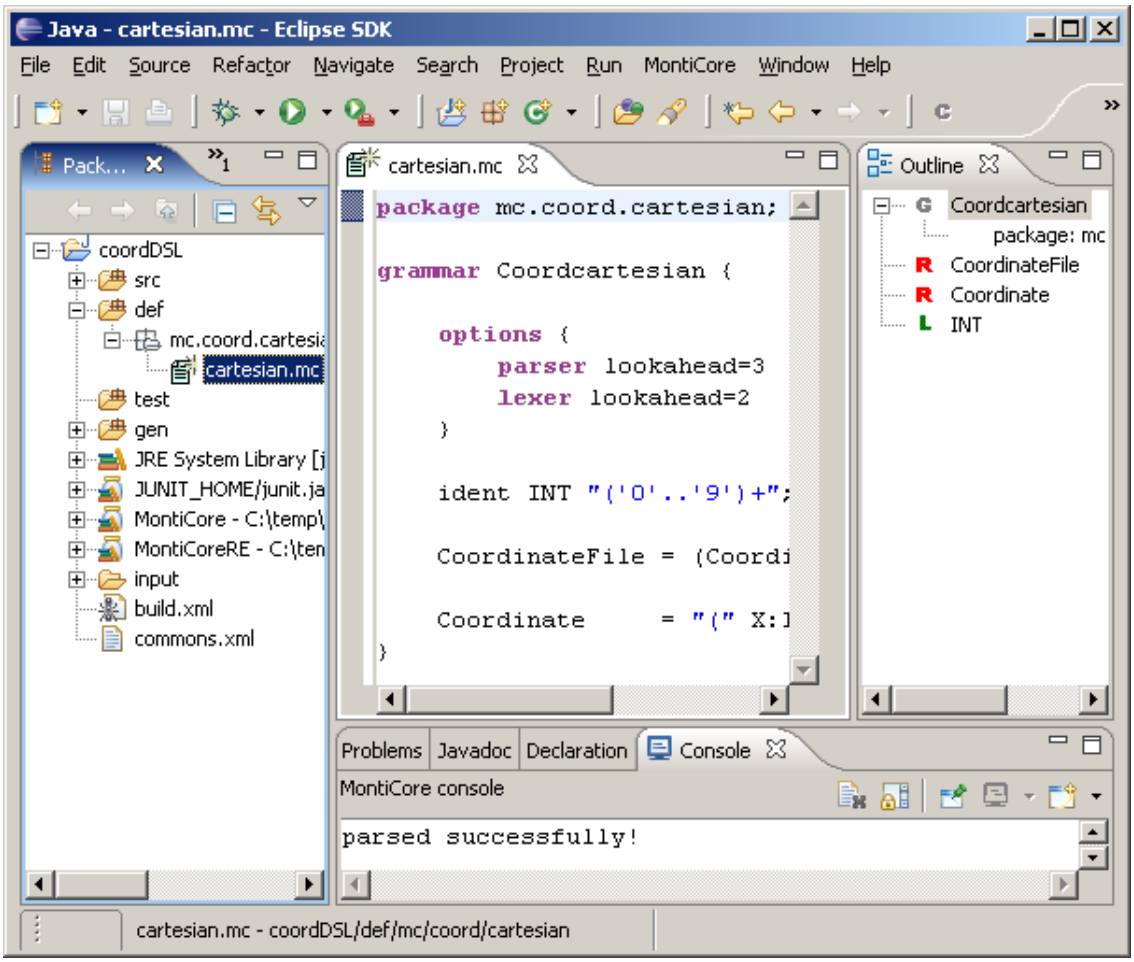


Abbildung 2.7.: Definition der CoordCartesian DSL in Eclipse

Beim Schreiben der Grammatik unter Eclipse können wir einige Komfortfunktionen nutzen, die uns das Plugin bietet. Das Syntaxhighlighting hebt Schlüsselwörter der

MontiCore-Grammatikbeschreibungssprache und Ausdrücke in Anführungszeichen hervor und erleichtert so die Lesbarkeit. Sobald wir eine .mc-Datei abspeichern, versucht MontiCore diese zu parsen und zeigt mögliche Fehler auf. So wurde in Abbildung 2.8 das abschließende Semikolon vergessen. Ist alles in Ordnung, erscheint in der Console „parsed successfully!“ und die sogenannte Outline wird erzeugt. Dabei handelt es sich um eine Übersicht der Definitionen unserer Sprache, die darüber hinaus die Navigation innerhalb der Grammatik erleichtert, indem wir durch einfaches Anklicken eines Elementes in der Outline zu der entsprechenden Definition in der Grammatik gelangen können.

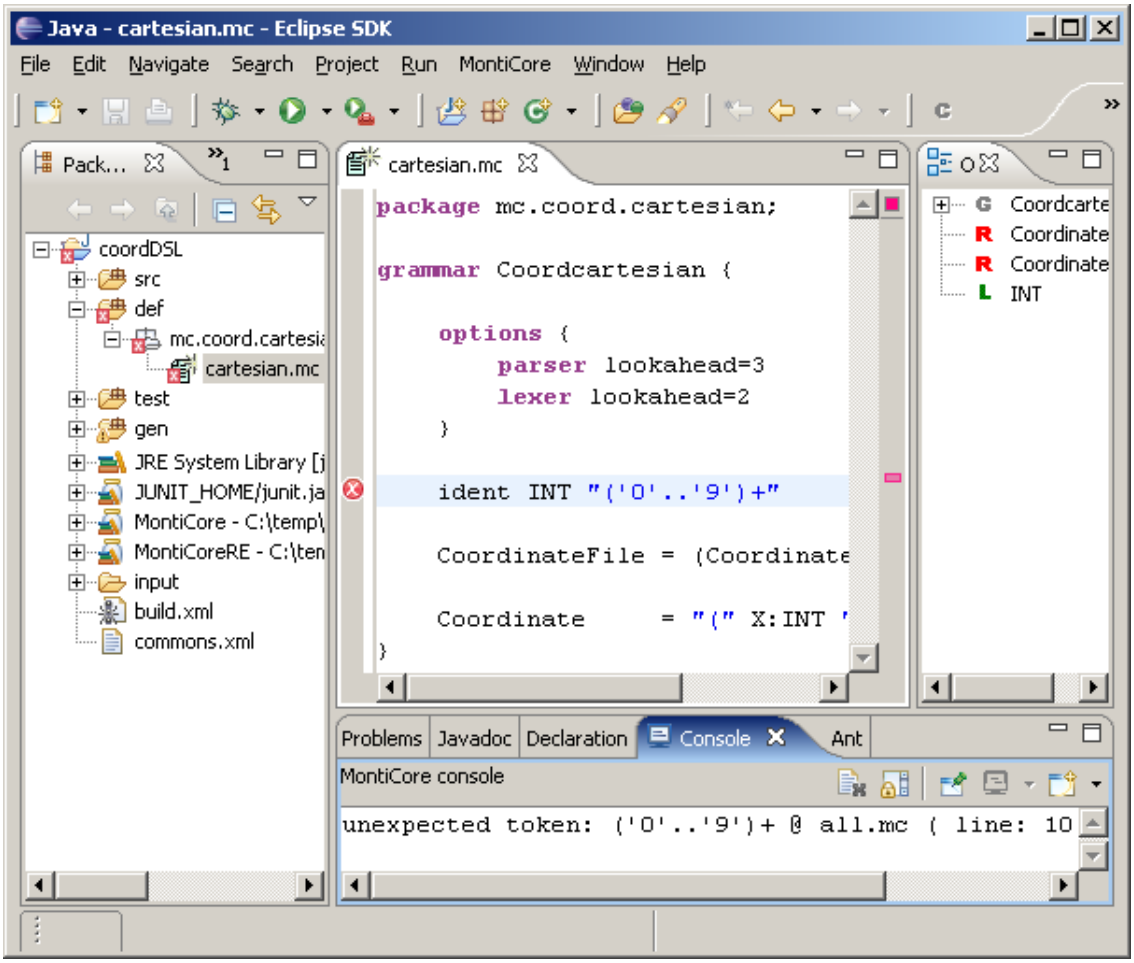


Abbildung 2.8.: Fehleranzeige bei der Grammatikdefinition in Eclipse

2.2.4. Generierung der Komponenten zur DSL-Verarbeitung

Aus der im vorigen Abschnitt beschriebenen Grammatik können wir nun automatisch verschiedene Komponenten zur Verarbeitung der DSL generieren. Das sind Lexer, Parser und AST-Klassen sowie weitere Dateien zu Dokumentationszwecken. Eine genaue Auflistung der generierten Dateien und ihrer Funktionen findet sich auf Seite 15 dieses Abschnitts.

Für die Generierung stehen unter Eclipse drei verschiedene Wege zur Verfügung, die alle zum selben Ergebnis führen, aber unterschiedliche Komfortansprüche erfüllen:

- Ausführen von `Generate.java` im Default Package von `src`. Hierbei handelt es sich um die einfachste Variante, die auch ohne die Infrastruktur von Eclipse möglich ist.
- Generierung über das `compile`-Target der Ant-Datei `build.xml` (siehe Abschnitt 2.2.1).
- Anschalten der `MontiCore Nature` (Rechtsklick auf das Projekt > Enable MontiCore Nature). Dann wird direkt nach dem Abspeichern von Änderungen in der Grammatikdefinition der Generierungsprozess automatisch gestartet.

Eine vierte Variante ist der Aufruf der jar-Datei von MontiCore über die Kommandozeile. Diese ist in Abschnitt 2.3 beschrieben.

Wie schon in Abschnitt 2.2.1 erwähnt, bietet die Nutzung der Ant-Datei neben der Generierung weitere Möglichkeiten, weshalb wir in diesem Tutorial die `build.xml` verwenden wollen. Wir öffnen also die Ant-Ansicht über „Window > Show View > Ant“ und ziehen die xml-Datei auf das neue Fenster. Durch einfachen Doppelklick auf das Target `compile` starten wir den Generierungsprozess (siehe Abbildung 2.9).

Bei der Generierung wird die Grammatik von MontiCore eingelesen und ANTLR-Code sowie die passenden AST-Klassen und eine API zum Aufbau eines ASTs erzeugt. Der AST steht für die entstehende Datenstruktur, die ein Speicherabbild der eingelesenen Datei darstellt.

Schließlich werden aus dem ANTLR-Code Parser und Lexer für die DSL erzeugt. Diese sind in der Lage, einen getypten AST mittels der generierten AST-Klassen aufzubauen. Dazu nutzt MontiCore die gut integrierten, aber auch einzeln einsetzbaren Beschreibungssprachen für MontiCore-Grammatik und -Klassendefinition (siehe Kapitel 4 und 5).

Die generierten Dateien werden im Ordner `gen` abgelegt. Für die hier vorgestellte Koordinaten-DSL sind dies folgende Dateien im Package `mc.coord.cartesian`:

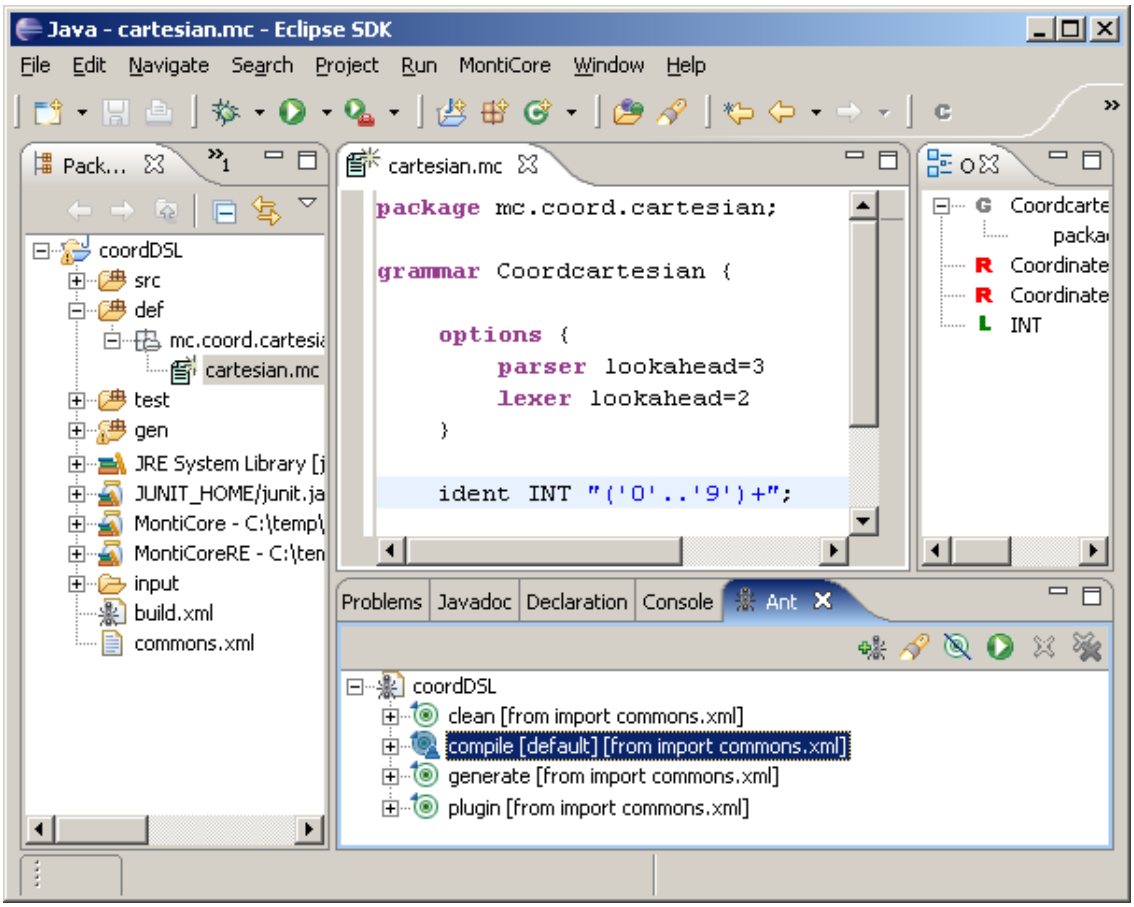


Abbildung 2.9.: Generierung der Infrastruktur zu einer Grammatik über das Ant-Target `compile` der `build.xml`

AST-Klassen:

- `ASTCoordinateFile.java`:
  Diese AST-Klasse entspricht der ersten Regel (Zeile 12 in Abbildung 2.6) der Grammatik. Sie enthält eine Liste der Koordinaten über ein `ASTCoordinateList`-Objekt.

- `ASTCoordinate.java`:
  Ein Objekt dieser Klasse repräsentiert eine Koordinate der DSL, abgeleitet aus der zweiten Grammatik-Regel (Zeile 14 in Abbildung 2.6).

- `ASTCoordinateList.java`:
  Eine Listenklasse für die Speicherung mehrerer Koordinaten vom Typ `ASTCoordinate`.

- `ASTConstantsCoordcartesian.java`:
  Definitionen von Konstantengruppen (siehe Abschnitt 3.2). Wurden in der Koordinaten-DSL nicht verwendet.

Lexer:

- `CoordcartesianLexer.java`:
  Der von ANTLR aus `Coordcartesian.g` erzeugte Lexer. Dieser wird von den Parsern instanziiert und braucht damit nicht weiter beachtet werden.
- `CoordcartesianParserTokenTypes.java`:
  Interface für den Zugriff auf alle für die Koordinaten-DSL gültigen Lexer-Tokens.
- `CoordcartesianParserTokenTypes.txt`:
  Definition einiger Standard-Tokens für den Lexer.
- `CommonTokenTypes.txt`:
  Definition der für die Koordinaten-DSL gültigen Lexer-Tokens.

Parser:

- `CoordcartesianCoordinateFileMCConcreteParser.java`:
  Parser für die erste Regel der Grammatik (Zeile 12 in Abbildung 2.6).
- `CoordcartesianCoordinateMCConcreteParser.java`:
  Parser für die zweite Regel der Grammatik (Zeile 14 in Abbildung 2.6).
- `CoordcartesianParser.java`:
  Der von ANTLR aus `Coordcartesian.g` erzeugte Parser. Dieser wird von den Parsern instanziiert und braucht damit nicht weiter beachtet werden.

Dokumentation:

- `cd.dot`:
  GraphViz-Datei für die Erzeugung einer graphischen Darstellung des Koordinaten-ASTs als UML-Klassendiagramm. GraphViz ist eine Open Source Visualisierungssoftware für Graphen und kann von [WWWn] heruntergeladen werden. Die entsprechende Postscript-Datei `cd.ps` mit der Darstellung des Klassendiagramms wird nur erzeugt, wenn GraphViz installiert ist und im Pfad des Betriebssystems aufgenommen ist (siehe Abschnitt 2.1).

- `Coordcartesian.ebnf`:
  Die Koordinaten-DSL als EBNF (Extended Backus-Naur Form). Diese bietet eine gute Übersicht und auch ein leichteres Verständnis der in Abbildung 2.6 vorgestellten Grammatik.

- `Coordcartesian.g`:
  Die ANTLR-Grammatikdatei.

Wie diese Infrastruktur genutzt werden kann, wird in den folgenden Abschnitten gezeigt. Doch zuvor wollen wir uns ansehen, wie wir Koordinaten unserer coordDSL einlesen können. Da Sprachen notwendigerweise vor der Verarbeitung zu lesen sind, stellt der Parser einen festen Bestandteil jedes DSL-Werkzeugs dar. Aus diesem Grund bietet MontiCore die Möglichkeit, diesen Arbeitsschritt als sogenannten Workflow (siehe Kapitel 3) direkt aus der Grammatik heraus zu erzeugen. Hierzu gibt es das Konzept `dsltool` (siehe Abschnitt 5.4). Ein Konzept ist ein Erweiterungspunkt einer Grammatik, um zusätzliche Eigenschaften oder Infrastruktur der dazugehörigen Sprache zu definieren. In Abbildung 2.10 ist die um das dsltool-Konzept erweiterte Gammatik unserer Koordinaten-DSL zu sehen. Dabei wurden folgende Elemente definiert:

- Zeile 11 definiert `CoordinateFile` als Startregel unserer DSL. `CartesianRoot` bezeichnet dabei den Typ des Objekts, das die Eingabedatei für die Weiterverarbeitung repräsentiert.

- Um `CartesianRoot`-Objekte für eine Eingabedatei zu erzeugen, spezifiziert Zeile 13 eine Fabrik `cartesianfactory` vom dafür eigens generierten Typ `CartesianRootFactory`. Diese instanziiert auch den zugehörigen Parser. Da hier mehrere Parser für eingebettete Sprachen kombiniert werden können, muss neben der Angabe des vollqualifizierten Namen für die Startregel auch ein Parser mit dem Stereotypen `«start»` gekennzeichnet werden. Dieser dient als Start-Parser, in den die anderen Parser eingebettet werden.

- Der eigentliche Parse-Workflow wird in Zeile 18 auf `CoordinateFile` als Ausgangspunkt definiert.

Nach der Generierung finden wir unter `gen/mc/coord/cartesian/` die drei Dateien `CartesianParsingWorkflow.java`, `CartesianRoot.java` und `CartesianRootFactory.java`, die die oben beschriebene Funktionalität beinhalten.

Schließlich müssen wir den Parse-Workflow noch aufrufen. Da wir nun dabei sind, ein Werkzeug zur Verarbeitung einer DSL zu schreiben, erstellen wir die ausführbare Klasse `CoordTool.java` im Package `mc.coord` unter `src` und leiten diese von `DSLTool` ab (siehe Kapitel 3). Diese ist dabei in zwei Teile gegliedert (siehe Abbildung 2.11):

```
1  package mc.coord.cartesian;
2
3  grammar Coordcartesian {
4
5      options {
6          parser lookahead=3
7          lexer lookahead=2
8      }
9
10     concept dsltool {
11         root CartesianRoot<CoordinateFile>;
12
13         rootfactory CartesianRootFactory for CartesianRoot<CoordinateFile> {
14             mc.coord.cartesian.Coordcartesian.CoordinateFile
15                 cartesianfactory <<start>>;
16         }
17
18         parsingworkflow CartesianParsingWorkflow for
19             CartesianRoot<CoordinateFile>;
20     }
21
22     ident INT "('0'..'9')+";
23
24     CoordinateFile = (Coordinates:Coordinate)+;
25
26     Coordinate     = "(" X:INT "," Y:INT ")";
27 }
```

Abbildung 2.10.: cartesian.mc - Grammatik der CoordCartesian DSL mit Startregel und Parse-Workflow-Definition

- In der `main`-Methode spezifizieren wir, wie sich unser Werkzeug beim Aufruf verhalten soll. Dies wollen wir zusätzlich über Parameter steuern können. Deshalb erzeugen wir als erstes eine `CoordTool`-Instanz und fügen die beim Aufruf der Methode übergebenen Parameter hinzu (Zeilen 9 - 14 in Abbildung 2.11). Über den Parameter `-parse` (siehe Abschnitt 2.3) kann man etwa angeben, welche Parse-Workflows ausgeführt werden sollen. Wenn kein Workflow angegeben wurde, wollen wir einfach alle Workflows ausführen, die in der `init`-Methode mit "`parse`" gekennzeichnet wurden (Zeilen 16 - 19). Als default-Verzeichnis für die Ausgabe unserer Workflows definieren wir in Zeile 22 das in Abschnitt 2.2.1 zu diesem Zweck angelegte Verzeichnis `output`. Die Klasse MCG steht für "Monticore Globals" und enthält Konstanten rund um den Generierungsprozess. Schließlich rufen wir die `init`- und die `run`-Methode des `CoordTool`s auf (Zeile 25 und 26).

- Die zur Verfügung stehenden Workflows werden in der `init`-Methode bekanntgegeben. Um Fehlermeldungen auf der Konsole ausgeben zu können, wird zu-

erst ein Error-Handler initialisiert. Über eine `DSLRootFactoryByFileExtension` können wir einzulesende Dateien anhand ihrer Dateiendung unterscheiden (Zeile 34 - 36). Zurzeit haben wir nur die Endung `cart` für die kartesischen Koordinaten. Diese fügen wir der Factory hinzu (Zeile 38 - 40) und setzen die Factory als RootFactory unserer Sprache ein. Schließlich fehlt noch der Workflow. Dieser wird dem Konfigurationsobjekt `configuration` des DSL-Tools hinzugefügt, nachdem die Root-Klasse dem Konfigurationsobjekt bekanntgegeben wurde (Zeile 44 - 51). Über das Schlüsselwort "`parse`" kann der Workflow genutzt werden (vgl. Zeile 18).

Wie wir im Folgenden sehen werden, bietet die hier vorgestellte Form der Einbindung der Workflows Flexibilität für Erweiterungen unserer DSL. Unser Werkzeug ist nun vorbereitet, um über einen Kommandozeilenaufruf gestartet zu werden (siehe Abschnitt 2.3). Da wir aber hier unter Eclipse arbeiten, schreiben wir uns eine weitere kleine Klasse `src/mc/coord/Run.java`, die diesen Aufruf übernimmt (siehe Abbildung 2.12). Dabei geben wir als Parameter nur das input-Verzeichnis an, um unser Beipiel `input/coordinates.cart` schließlich zu parsen. Die erlaubten Parameter entsprechen den Parametern, die auch auf der Kommandozeile angegeben werden können (siehe Abbildung 2.29 in Abschnitt 2.3). Den Parse-Workflow haben wir in unserem `CoordTool` als default angegeben.

Beim Aufruf von `Run.java` über „Run > Run As > Java Application“ wird nun die Datei `input/coordinates.cart` geparst und in den AST überführt. Da jedoch noch keine Funktionalität angefügt ist, gibt es noch keine Ausgabe.

2.2.5. Programmierung eines Pretty-Printers

Eine einfache Art und Weise zu kontrollieren, ob das Parsen erfolgreich war, ist die Ausgabe des geparsten ASTs. Diese als Pretty-Printer bezeichnete Funktionalität kann auch später z.B. bei Transformationen für die Ausgabe des ASTs genutzt werden (siehe Abschnitt 2.2.6) und ist für fast alle DSLs sinnvoll. MontiCore enthält daher die Klasse `ConcretePrettyPrinter`, von der wir unseren Pretty-Printer ableiten. Wir legen die Implementierung in der Klasse `CartesianConcretePrettyPrinter.java` im neuen Package `mc.coord.cartesian.prettyprint` unter `src` ab (siehe Abbildung 2.13). Unser Pretty-Printer redefiniert zwei Methoden von `ConcretePrettyPrinter`:

- `getResponsibleClasses()`, für die Ausgabe der von unserem Pretty-Printer behandelten AST-Klassen und
- `prettyPrint()`, für die Ausgabe des ASTs.

```
1  package mc.coord;
2
3  import mc.ConsoleErrorHandler; ...
4
5  public class CoordTool extends DSLTool {
6
7      public static void main(String[] args) {
8
9          CoordTool m = new CoordTool();
10
11         // Determine parameters from command line parameters
12         Parameters parameters =
13             HandleParams.handle(args, m.getModelInfrastructureProvider());
14         m.setParameters(parameters);
15
16         // Default ExecutionUnits
17         if (parameters.getParses().isEmpty()) {
18             parameters.addParse(Parameters.ALL, "parse");
19         }
20
21         //Default output directory:
22         MCG.OUTPUT_DIR="output";
23
24         // init and run
25         m.init();
26         m.run();
27     }
28
29     private void init() {
30
31         // Report errors to the console
32         addErrorHandler(new ConsoleErrorHandler());
33
34         // Determine file type by file extension
35         DSLRootFactoryByFileExtension rootfactory =
36             new DSLRootFactoryByFileExtension(this);
37
38         // Cartesian coordinates end with file extension "cart"
39         CartesianRootFactory cart = new CartesianRootFactory(this);
40         rootfactory.addFileExtension("cart", cart);
41
42         setDSLRootFactory(rootfactory);
43
44         // Add workflows to the configuration
45         configuration = new DSLToolConfiguration();
46
47         configuration.addDSLRootClassForUserName("cartesian",
48                                                  CartesianRoot.class);
49
50         configuration.addExecutionUnit("parse",
51                 new CartesianParsingWorkflow(CartesianRoot.class));
52     }
53 }
```

Abbildung 2.11.: CoordTool.java - Tool-Konfiguration der coordDSL

```
1  package mc.coord;
2
3  public class Run {
4
5      public static void main(String[] args) {
6          CoordTool.main(new String[] {"input"});
7      }
8      //corresponds to command line:
9      //    java -classpath de.monticore.re_1.0.0.jar;build
10     //         mc.coord.CoordTool input
11 }
```

Abbildung 2.12.: Run.java - Ausführbare Java-Klasse für den Aufruf des DSL-Tools

```
1  package mc.coord.cartesian.prettyprint;
2
3  import mc.ast.ASTNode; ...
4
5  public class CartesianConcretePrettyPrinter extends ConcretePrettyPrinter {
6
7      public Class[] getResponsibleClasses() {
8          return new Class[] {ASTCoordinateFile.class, ASTCoordinate.class};
9      }
10
11     public void prettyPrint(ASTNode a, IndentPrinter printer) {
12         Visitor.run(new CartesianPrettyPrinterConcreteVisitor(printer), a);
13     }
14 }
```

Abbildung 2.13.: CartesianConcretePrettyPrinter.java - Implementierung des Pretty-Printers

Die Ausgabe des ASTs setzen wir über das Visitor-Muster um (zur Erläuterung der Grundidee dieses Musters siehe [GHJV96]). Hierzu erstellen wir gemäß Abbildung 2.14 eine neue Subklasse `CartesianPrettyPrinterConcreteVisitor` von `ConcreteVisitor` im Package `mc.coord.cartesian.prettyprint`. Die visit-Methode in Zeile 13 gibt dabei die Koordinaten über den bei der Instanziierung übergebenen `IndentPrinter` aus, wenn ein Knoten vom Typ `ASTCoordinate` im AST durchlaufen wird.

Im vorigen Abschnitt haben wir mit Unterstützung von MontiCore einen Workflow für das Parsen von Eingabedateien erzeugt. Wie alle Erweiterungen unserer DSL, wollen wir auch den Pretty-Printer als Workflow einbinden. Da sich ein Workflow aus mehreren Teil-Workflows zusammensetzen kann, können wir auf diese Art und Weise komplexe Arbeitsschritte aufbauen. Die hier von Hand codierte Implementierung des Pretty-Print-Workflows zeigt Abbildung 2.15.

```
1  package mc.coord.cartesian.prettyprint;
2
3  import mc.ast.ConcreteVisitor; ...
4
5  public class CartesianPrettyPrinterConcreteVisitor extends ConcreteVisitor {
6
7      private IndentPrinter p;
8
9      public CartesianPrettyPrinterConcreteVisitor(IndentPrinter printer) {
10         this.p = printer;
11     }
12
13     public void visit(ASTCoordinate a) {
14         p.print("("+a.getX()+","+a.getY()+") ");
15     }
16 }
```

Abbildung 2.14.: CartesianPrettyPrinterConcreteVisitor.java - Visitorimplementierung des Pretty-Printers für kartesische Koordinaten

Um einen von der abstrakten generischen Klasse `DSLWorkflow` abgeleiteten Workflow zu definieren, muss die Methode `run()` implementiert werden. Unser Pretty-Print-Workflow soll dabei ein `CartesianRoot`-Objekt bearbeiten, das beim Aufruf übergeben wird (Zeile 5 und 12 in Abbildung 2.15). Dieses Root-Objekt enthält auch den AST (Zeile 14). Nun brauchen wir nur noch unseren Pretty-Printer instanziieren, eine neue Datei für die Ausgabe dem Root-Objekt hinzufügen und die `prettyPrint`-Methode aufrufen (Zeile 16 - 30). Darüber hinaus bekommt ein Workflow bei der Instanziierung noch die Klasse des Root-Objektes übergeben, für die er zuständig ist (Zeile 7 - 9).

Schließlich muss der neue Workflow wieder unserem `CoordTool` bekannt gegeben werden. Wir wollen diesen ausführen, wenn kein anderer Workflow spezifiziert wurde (siehe Abbildung 2.16). Diese Zeilen fügen wir nach den default-Parse-Workflow in Zeile 17 von `CoordTool.java` (Abbildung 2.11) hinzu. Außerdem müssen wir noch den Workflow in der `init`-Methode dem `configuration`-Objekt bekanntgeben, wie Abbildung 2.17 zeigt.

Dank unserer default-Konfiguration des Parse-Workflows aus Abschnitt 2.2.4 und des Pretty-Print-Workflows, brauchen wir keinen von beiden beim Aufruf mit anzugeben. Auch für die Ausgabe haben wir mit `output` ein Standard-Verzeichnis in `CoordTool.java` angegeben (siehe Abbildung 2.11). Führen wir nun die in Abschnitt 2.2.4 erstellte `Run.java` aus, finden wir die Pretty-Print-Ausgabe unserer Beispielkoordinaten in `output/coordinates.cart`.

```
1  package mc.coord.cartesian.workflows;
2
3  import java.io.File; ...
4
5  public class PrettyPrintWorkflow extends DSLWorkflow<CartesianRoot> {
6
7      public PrettyPrintWorkflow(Class<CartesianRoot> responsibleClass) {
8          super(responsibleClass);
9      }
10
11     @Override
12     public void run(CartesianRoot dslroot) {
13
14         ASTCoordinateFile ast = dslroot.getAst();
15
16         //Create PrettyPrinter
17         PrettyPrinter p = new PrettyPrinter();
18         p.addConcretePrettyPrinter(new CartesianConcretePrettyPrinter());
19
20         //Get name for output file
21         String name = new File(dslroot.getFilename()).getName();
22
23         //Create file
24         GeneratedFile f = new GeneratedFile("", name, WriteTime.IMMEDIATELY);
25
26         //Register it for writing
27         dslroot.addFile(f);
28
29         // Pretty-print the cartesian coordinates
30         p.prettyPrint(ast, f.getContent());
31     }
32 }
```

Abbildung 2.15.: PrettyPrintWorkflow.java - Workflow für die Wiederausgabe von kartesischen Koordinaten

```
1 //To be added in CoordTool.java at line 20
2 if (parameters.getExecutionsUnits().isEmpty()) {
3     parameters.addExecutionUnit("cartesian", "prettyprint");
4 }
```

Abbildung 2.16.: Einbinden des Pretty-Print-Workflows in `CoordTool.java`

2.2.6. Intra-Modelltransformation

Bei einer Transformation werden Daten oder Strukturen in ein anderes Format überführt, wobei sich dieses Format nicht notwendigerweise vom Ursprungsformat unterscheiden muss. In Abschnitt 2.2.5 haben wir die generierten AST-Klassen für die

```
1 //To be added in CoordTool.java at the end of the init()-method
2 configuration.addExecutionUnit("prettyprint", new
3     mc.coord.cartesian.workflows.PrettyPrintWorkflow(CartesianRoot.class));
```

Abbildung 2.17.: Initialisierung des Pretty-Print-Workflows in `CoordTool.java`

Erstellung eines Pretty-Printers genutzt. Dabei haben wir den AST in eine textuelle Form überführt und damit schon unsere erste Transformation erstellt.

Durch die Definition mehrerer DSLs werden über MontiCore die jeweils generierten AST-Klassen und ihre API verfügbar gemacht. Daher ist es möglich, nach dem Visitor-Muster [GHJV96] einen solchen zu erstellen, der über eine konkrete AST-Instanz hinwegläuft und die API der Zielsprache benutzt, um einen zweiten AST aufzubauen. Auf diese Weise übersetzen wir ein Modell, nämlich unsere DSL, in ein anderes. Die Modelltransformation wird durch den Aufbau des ASTs der Zielsprache realisiert. Codegenerierung ist eine spezielle Form der Modelltransformation, bei der der erzeugte AST Klassenstrukturen repräsentiert, die dann über einen PrettyPrint-Visitor ausgegeben werden.

MontiCore verwendet eine erweiterte Variante des Visitor-Musters. Ein Hauptvisitor `mc.ast.Visitor` ist dabei im Stande, mehrere Client-Visitoren zu verwalten. Diese müssen von `ConcreteVisitor` abgeleitet sein, um vom Hauptvisitor angenommen werden zu können. Dieses Verfahren erlaubt die Verwendung mehrerer für bestimmte AST-Knoten spezialisierte Visitoren für einen AST. Dadurch kann ein AST dynamisch um neue AST-Knoten erweitert werden, ohne dass ursprüngliche Visitorimplementierungen angepasst werden müssen.

Eine Intra-Modelltransformation ist eine Transformation, bei der die abstrakte Syntax des Ursprungs- und Zielmodell gleich sind. Ein einfaches Beispiel in Bezug auf die Koordinaten-DSL ist eine Spiegelung an der Ursprungsgeraden, d.h. eine Vertauschung der x- und y-Koordinate. Die Implementierung einer solchen Transformation wird in Abbildung 2.18 demonstriert und wir legen diese unter `mc.coord.cartesian` ab.

Die entsprechende Workflow-Implementierung wird in Abbildung 2.19 gezeigt. Bis zu diesem Punkt spiegelt unser Workflow die Koordinaten nur auf der AST-Ebene. Da wir die gespiegelten Koordinaten jedoch auch ausgeben wollen, können wir hier unseren Pretty-Print-Workflow aus dem vorigen Abschnitt wiederverwenden, indem wir einen Composite-Workflow wie in Abbildung 2.20 erstellen.

Im Gegensatz zu den bisherigen Workflows wollen wir den Mirror-Workflow nur dann ausführen, wenn dies explizit über die Aufrufparameter gewünscht ist. Deshalb fügen wir nur die Zeile aus Abbildung 2.21 an das Ende der `init`-Methode von `CoordTool.java` ein. Wollen wir nun die Koordinaten unserer Beispieldatei spiegeln, müssen wir den Schalter `-workflow` nutzen, dem als erster Parameter die

```
1  package mc.coord.cartesian;
2
3  import mc.ast.ConcreteVisitor;
4
5  public class Mirror extends ConcreteVisitor {
6
7      public void visit(ASTCoordinate a) {
8          String y = a.getY();
9          a.setY(a.getX());
10         a.setX(y);
11     }
12 }
```

Abbildung 2.18.: Mirror.java - Visitor zur Spiegelung der Koordinaten an der Ursprungsgerade

```
1  package mc.coord.cartesian.workflows;
2
3  import mc.DSLWorkflow; ...
4
5  public class MirrorWorkflow extends DSLWorkflow<CartesianRoot> {
6
7      public MirrorWorkflow(Class<CartesianRoot> responsibleClass) {
8          super(responsibleClass);
9      }
10
11     @Override
12     public void run(CartesianRoot dslroot) {
13         Visitor.run(new Mirror(), dslroot.getAst());
14     }
15 }
```

Abbildung 2.19.: MirrorWorkflow.java - Workflow zur Spiegelung der Koordinaten an der Ursprungsgerade

Bezeichnung der Root-Klasse folgt, auf den der Workflow angewendet werden soll (in diesem Fall `cartesian`; siehe Abbildung 2.11, Zeile 47) und als zweiter Parameter die Bezeichnung des Workflows aus Abbildung 2.21. Ausserdem geben wir ein Ausgabeverzeichnisses über den Parameter „-o“ an. Den vollständigen Aufruf zeigt Abbildung 2.22.

2.2.7. Inter-Modelltransformation

Als Beispiel für eine Transformation zwischen zwei verschiedenen Modellen setzen wir hier die kartesischen Koordinaten in Polarkoordinaten um. Dazu beschreiben wir zunächst das Zielmodell als DSL. Wir wenden also genau dieselben Mechanismen

```
1  package mc.coord.cartesian.workflows;
2
3  import mc.DSLCompositeWorkflow; ...
4
5  public class PrintMirrorWorkflow extends DSLCompositeWorkflow<CartesianRoot,
6                                           DSLWorkflow<CartesianRoot>> {
7
8      public PrintMirrorWorkflow() {
9          super(responsibleClass);
10         addWorkflowComponent(new MirrorWorkflow());
11         addWorkflowComponent(new PrettyPrintWorkflow());
12     }
13 }
```

Abbildung 2.20.: PrintMirrorWorkflow.java - Composite-Workflow für die Ausgabe der gespiegelten Koordinaten

```
1 //To be added in CoordTool.java at the end of the init()-method
2 configuration.addExecutionUnit("printmirror",
3         new PrintMirrorWorkflow(CartesianRoot.class));
```

Abbildung 2.21.: Einbindung des Mirror-Workflows in die `init`-Methode von `CoordTool.java`

```
1 CoordTool.main(new String[] {"-o","output/mirror","input",
2                              "-workflow","cartesian","printmirror"});
3 //corresponds to command line:
4 //    java -classpath de.monticore.re_1.0.0.jar;build
5 //         mc.coord.CoordTool -o output/mirror input
6 //                            -workflow cartesian printmirror
```

Abbildung 2.22.: Expliziter Aufruf des Mirror-Workflows in `Run.java`

an, wie für die erste Koordinaten-DSL. Die Grammatik ist in Abbildung 2.23 zu sehen. Da die Grammatiken und damit die generierten AST-Klassen in verschiedenen Packages liegen, können wir die Nichtterminale analog zur Coordcartesian-DSL wählen.

Wie in den vorigen Abschnitten beschrieben, können wir auch für diese DSL einen Pretty-Printer schreiben und zusammen mit dem Parse-Workflow in der `CoordTool.java` initialisieren. Zusätzlich wollen wir die Dateiendung „polar“ für Eingabedateien von Polarkoordinaten spezifizieren. Dies geschieht analog zur cartesian-DSL aus Abbildung 2.11.

Die Transformation von einem Modell in ein anderes erfolgt am einfachsten durch Implementierung eines Visitors. Dieser enthält ein Attribut `result`, in der das Er-

```
1  package mc.coord.polar;
2
3  grammar Coordpolar {
4
5      options {
6          parser lookahead=1
7          lexer lookahead=2
8      }
9
10     concept dsltool {
11         root PolarRoot<CoordinateFile>;
12
13         rootfactory PolarRootFactory for PolarRoot<CoordinateFile> {
14             mc.coord.polar.Coordpolar.CoordinateFile
15                 polarfactory <<start>>;
16         }
17
18         parsingworkflow PolarParsingWorkflow for
19             PolarRoot<CoordinateFile>;
20     }
21
22     ident REAL "('0'..'9')+(','('0'..'9')+)?";
23
24     CoordinateFile = (Coordinates:Coordinate)+;
25
26     Coordinate     = "[" D:REAL ";" Phi:REAL "]";
27 }
```

Abbildung 2.23.: polar.mc - Grammatik für Polarkoordinaten

gebnis der Transformation als AST des Zielmodells abgelegt wird. Die Transformation selbst wird durch die `visit`-Methoden durchgeführt. Die vollständige Implementierung zeigt Abbildung 2.24.

Die Implementierung als Workflow ist in Abbildung 2.25 zu sehen und unterscheidet sich von den bisherigen Workflows nur in dem Punkt, dass durch den Visitor ein neuer AST aufgebaut wird, anstatt einen bestehenden zu verändern. Die Einbindung in `CoordTool.java`, sowie der Aufruf erfolgen analog zu den vorigen Abschnitten.

Das vollständige Beispiel der Koordinaten-DSL steht beim Anlegen eines neuen MontiCore-Projektes zur Verfügung, wenn nach der Vergabe eines Projektnamens „next“ anstelle von „finish“ ausgewählt wird (siehe Abschnitt 2.2.1). Diese muss für die Verwendung nur noch über die `build.xml` kompiliert werden.

```
1  package mc.coord.transform;
2
3  import java.text.DecimalFormat; ...
4
5  public class CartesianToPolar extends ConcreteVisitor {
6
7      protected mc.coord.polar.ASTCoordinateFile result;
8
9      public mc.coord.polar.ASTCoordinateFile getResult() {
10         return result;
11     }
12
13     public void visit(mc.coord.cartesian.ASTCoordinateFile a) {
14         result = new mc.coord.polar.ASTCoordinateFile();
15     }
16
17     public void visit(mc.coord.cartesian.ASTCoordinate a) {
18
19         DecimalFormat Reals = new DecimalFormat("0.000");
20
21         // d = sqrt(x*x + y*y)
22         String d = Reals.format(
23                 Math.sqrt(Double.parseDouble(a.getX())
24                 * Double.parseDouble(a.getX())
25                 + Double.parseDouble(a.getY())
26                 * Double.parseDouble(a.getY())));
27
28         // angle = atan2(y,x)
29         String angle = Reals.format(
30                 Math.atan2(Double.parseDouble(a.getY()),
31                 Double.parseDouble(a.getX())));
32
33         result.getCoordinates().add(
34                 new mc.coord.polar.ASTCoordinate(d, angle));
35     }
36 }
```

Abbildung 2.24.: CartesianToPolar.java - Visitor für die Transformation zwischen kartesischen - und Polarkoordinaten

2.3. Nutzung als Standalone-Werkzeug

Nach der in Abschnitt 2.1.1 beschriebenen Installation kann MontiCore auch als Standalone-Werkzeug genutzt werden. Das in diesem Kapitel vorgestellte Tutorial ist so auch ohne Eclipse nachzuvollziehen.

Zuerst legen wir ein leeres MontiCore-Projekt `coordDSL` mit

```
java -jar de.monticore.examples_1.0.0.jar empty coordDSL
```

```
1  package mc.coord.transform;
2
3  import java.io.File; ...
4
5  public class CartesianToPolarWorkflow extends DSLWorkflow<CartesianRoot> {
6
7      public CartesianToPolarWorkflow(Class<CartesianRoot> responsibleClass) {
8          super(responsibleClass);
9      }
10
11     @Override
12     public void run(CartesianRoot dslroot) {
13
14         ASTCoordinateFile ast = dslroot.getAst();
15
16         //Transform cartesian to polar coordinates
17         CartesianToPolar transformer = new CartesianToPolar();
18         Visitor.run(transformer, ast);
19
20         //Create PrettyPrinter
21         PrettyPrinter p = new PrettyPrinter();
22         p.addConcretePrettyPrinter(new PolarConcretePrettyPrinter());
23
24         //Get name for output file
25         String name = new File(dslroot.getFilename()).getName();
26
27         //Create file
28         GeneratedFile f = new GeneratedFile("", name + ".polar",
29                                             WriteTime.IMMEDIATELY);
30
31         //Register it for writing
32         dslroot.addFile(f);
33
34         // Pretty-print the cartesian coordinates
35         p.prettyPrint(transformer.getResult(), f.getContent());
36     }
37
38 }
```

Abbildung 2.25.: CartesianToPolarWorkflow.java - Workflow für die Transformation zwischen kartesischen - und Polarkoordinaten

an. Der erzeugte Aufbau und Inhalt dieses Projektes entspricht der Beschreibung in Abschnitt 2.2.1.

Alle weiteren der in diesem Tutorial erarbeiteten Dateien können nun an entsprechenden Ordnern in dieser Verzeichnisstruktur abgelegt werden. Da sich die Implementierung der Grammatiken, Workflows, Visitoren und der sonstigen Sourcen nicht von der unter Eclipse unterscheidet, wird in diesem Abschnitt nicht näher darauf

eingegangen, sondern auf die Abschnitte 2.2.1 - 2.2.7 verwiesen.

Die Grammatik-Beschreibung aus Abbildung 2.10 auf Seite 19 kann mit MontiCore über folgenden Aufruf verarbeitet und die AST-Struktur damit generiert werden:

```
java -classpath de.monticore.re_1.0.0.jar;de.monticore_1.0.0.jar
mc.MontiCore def/mc/coord/cartesian/cartesian.mc
```

Abbildung 2.26 zeigt das Ergebnis eines erfolgreichen Aufrufs. In diesem Arbeitsschritt wird dabei ebenfalls von MontiCore die Infrastruktur für unsere DSL generiert, wie sie in Abschnitt 2.2.4 beschrieben wurde.

Abbildung 2.26.: Parsen der coordDSL über die Kommandozeile

Dieser Schritt kann automatisiert werden durch die Ant-Datei `build.xml`. In dieser müssen allerdings noch die Pfadangaben in den Eigenschaftswerten „mcgenerator“ und „mcruntime“, die auf de.monticore_1.0.0.jar bzw. de.monticore.re_1.0.0.jar verweisen, angepasst werden (siehe Abbildung 2.27). Danach können die Ant-Targets über `ant „Targetname“` aufgerufen werden. Die Generierung wird dementsprechend über

```
ant compile
```

gestartet, wie Abbildung 2.28 zeigt (siehe auch Abschnitt 2.2.1).

```
1  <project name="coordDSL" default="compile" basedir=".">
2  <description>
3      coordDSL
4  </description>
5  <property name="mcgenerator"
6            location="C:/coordDSL/de.monticore_1.0.0.jar" />
7  <property name="mcruntime"
8            location="C:/coordDSL/de.monticore.re_1.0.0.jar" />
9  <import file="commons.xml"/>
10 </project>
```

Abbildung 2.27.: Pfadanpassung in der `build.xml`

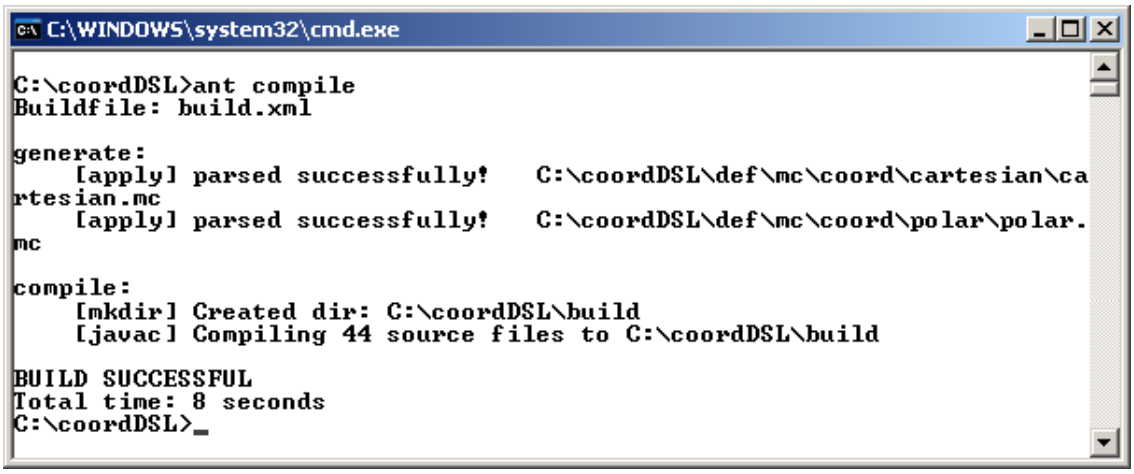


Abbildung 2.28.: Generierung der coordDSL-Infrastruktur mit Hilfe der `build.xml`

Bei der Nutzung der `build.xml` wird beim Target `compile` ebenfalls die Java-Dateien in `src` nach `build` compiliert, so dass anschließend z.B. der Mirror-Workflow unseres CoordTools aus Abbildung 2.21 über den Aufruf

```
java -classpath de.monticore.re_1.0.0.jar;build mc.coord.CoordTool
-o output/mirror input -workflow cartesian printmirror
```

gestartet werden kann (vergleiche Abbildung 2.22).

Die möglichen Parameter beim Aufruf eines unter MontiCore erstellten DSL-Tools werden durch

```
java -classpath de.monticore.re_1.0.0.jar;de.monticore_1.0.0.jar
mc.MontiCore -?
```

angezeigt (siehe Abbildung 2.29).

Das in diesem Kapitel vorgestellte Projekt ist in de.monticore.examples_1.0.0.jar enthalten und kann mit

```
java -jar de.monticore.examples_1.0.0.jar coord coordDSL
```

erzeugt werden.

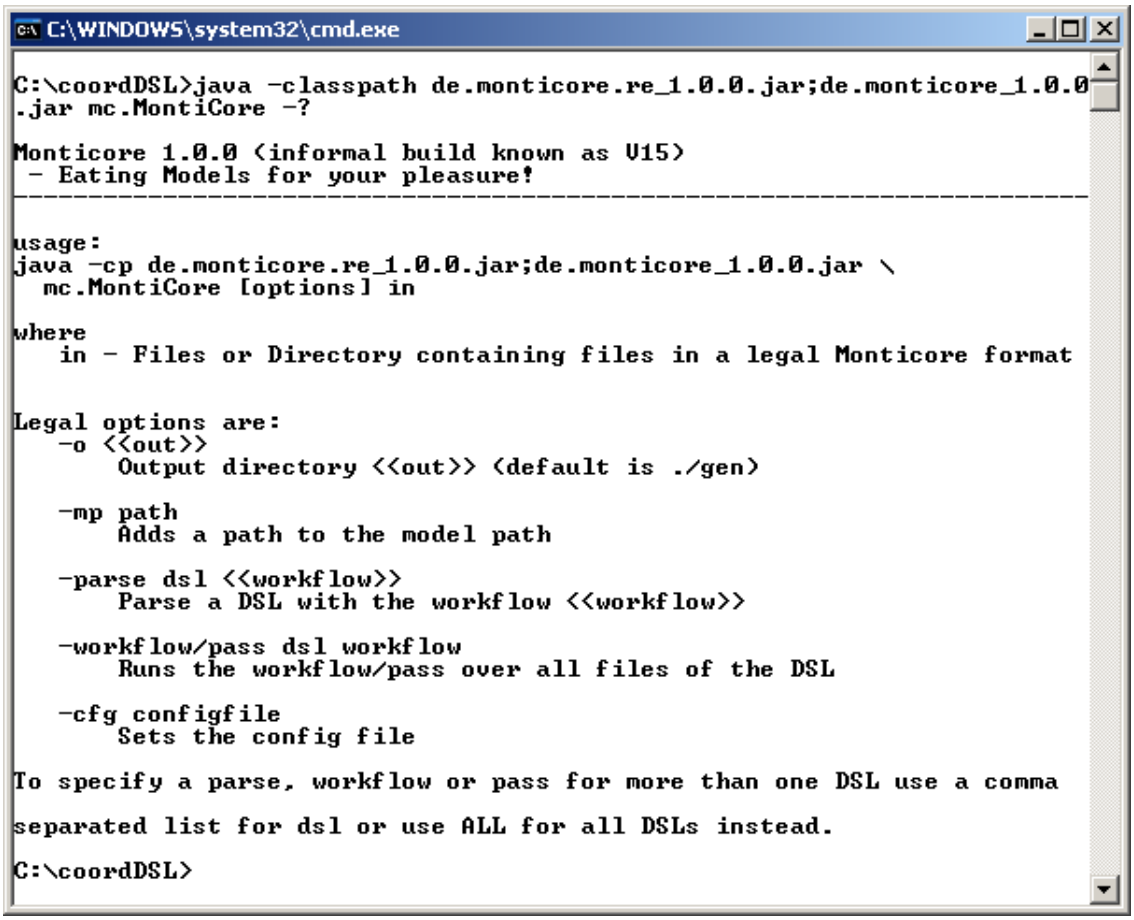


Abbildung 2.29.: Parameter für DSL-Tools

3. Softwareentwicklung mit MontiCore

Unter einem *Generator* verstehen wir eine Software, die aus einer Menge von Eingabedateien eine Menge von Ausgabedateien generiert. Die bekanntesten Beispiele für Generatoren sind Compiler, aber auch Software wie z.B. Javadoc [WWWo], Latex2HTML [WWWs] und Codegeneratoren für die UML fallen in diese Kategorie. Im Zusammenhang mit MontiCore sprechen wir von einem *Codegenerator*, wenn die Ausgabedateien aus Java-Quellcode bestehen. Die durch die Software durchgeführte Transformation lässt sich aus Abbildung 3.1 erkennen.

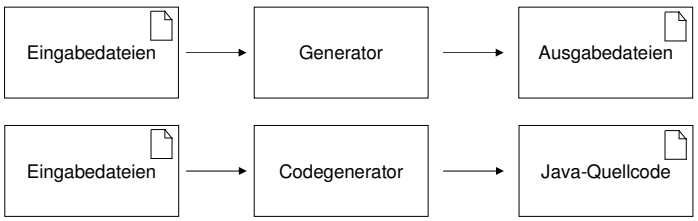


Abbildung 3.1.: Generatoren und Codegeneratoren

Unter „generative programming“ [CE00] wird ein Softwareentwicklungsparadigma verstanden, bei dem aus einer gegebenen Spezifikation ein angepasstes und optimiertes Zwischen- oder Endprodukt generiert wird. Dabei werden elementare und wiederverwendbare Implementierungen von Komponenten durch Konfiguration angepasst und miteinander verbunden.

MontiCore wird für zwei primäre Ziele entwickelt. Entsprechend Abbildung 3.1 kann MontiCore zur allgemeinen Bearbeitung von Dateien in der Informationsverarbeitung eingesetzt werden oder in der effizienten Entwicklung von Codegeneratoren für die Softwareentwicklung. In diesem Abschnitt werden die Komponenten von MontiCore und deren Zusammenwirken mit dem Ziel der Unterstützung im Softwareentwicklungsprozess diskutiert. Das führt zu einem mehrstufigen, später noch genauer zu behandelnden Generierungsprozess, vor allem weil erstens MontiCore selbst aus durch MontiCore generierte Komponenten besteht und zweitens ein Teil des Frameworks sowohl in MontiCore, als auch in dem von ihm generierten DSLTools genutzt werden kann.

MontiCore ist also ein Codegenerator, der sich zur Entwicklung komplexer Softwaresysteme einsetzen lässt. Es generiert dabei primär Komponenten, die die Grundlage für komplexere Generatoren bilden. Dazu werden meistens die generierten Komponenten in das MontiCore-DSLTool-Framework (vgl. Abschnitt 3.1) integriert und bilden dadurch einen Generator. Der Aufbau und die Entstehung ist in Abbildung 3.2 zu sehen. MontiCore generiert dabei aus der Sprachspezifikation angepasste Komponenten, die zur Sprachverarbeitung eingesetzt werden können. Diese können durch weitere handcodierte oder einer Bibliothek entnommene Komponenten ergänzt werden und bilden zusammen mit dem DSLTool-Framework einen neuen Generator. Da MontiCore hier teilweise Komponenten angepasst an eine konkrete Problemstellung generiert und zusammmen mit anderen Komponenten für eine spezielle Problemstellung konfiguriert, ist es ein konkretes Beispiel für das Paradigma „generative programming“.

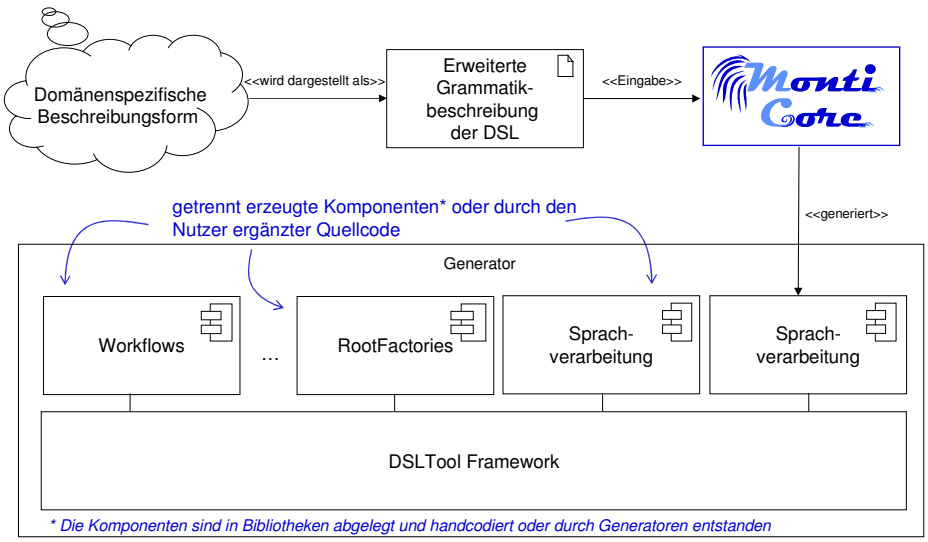


Abbildung 3.2.: DSLTool-Infrastruktur

Die von MontiCore generierten Komponenten sind in sich soweit abgeschlossen, dass sie auch innerhalb anderer Frameworks oder Applikationen zum Einsatz kommen können. Dieses erlaubt zukünftig etwa die Ausgestaltung eines Frameworks zur Analyse von Software-Projekten (AnalyseTool-Framework), das eine andere Ablauflogik und Erweiterungsmöglichkeiten bietet als ein DSLTool, jedoch die einzelne Komponenten einsetzen kann, die bereits im Zusammenhang mit dem DSLTool-Framework verwendet werden.

3.1. Aufbau eines DSLTools

Unter einem *DSLTool* verstehen wir einen Generator, der das MontiCore-DSLTool-Framework zur Ausführung verwendet. Ein DSLTool verarbeitet Eingabedokumente auf eine festgelegte Art und Weise und verwendet ein einheitliches durch das Framework zur Verfügung gestelltes System für Basisaufgaben wie Fehlermeldungen und Dateierzeugung.

Die zentralen Klassen des Frameworks sind aus Abbildung 3.3 ersichtlich. Dabei repräsentieren Objekte der Klasse DSLRoot einzelne Eingabedateien. Spezielle Unterklassen können Daten und Operationen enthalten, die nur für bestimmte Dateitypen relevant sind. In diesen Subklassen können gezielt zusätzliche Informationen abgelegt werden, die bei der Verarbeitung anfallen. Die DSLRoot-Objekte werden von einer DSLRootFactory instanziiert.

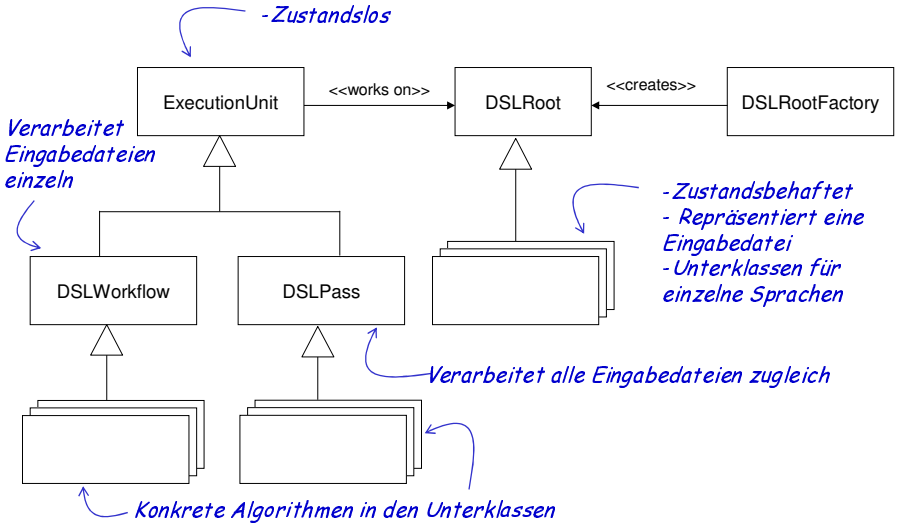


Abbildung 3.3.: Zentrale Klassen der DSLTool-Infrastruktur

Die Verarbeitung einer Datei (bzw. des entsprechenden DSLRoot-Objekts) geschieht durch Unterklassen der abstrakten Klasse ExecutionUnit. Dabei kann entweder die Klasse DSLPass überschrieben werden, falls alle Eingabedateien zur gleichen Zeit verarbeitet werden sollen. Dieses würde sich zum Beispiel bei der Erstellung eines Klassendiagramms aus einer Menge an Quellcodeeingabedateien anbieten. Alternativ wird eine Unterklasse von DSLWorkflow verwendet, falls die Verarbeitung der Datei einzeln erfolgen kann.

Ein DSLTool verarbeitet alle Dateien mit der in Abbildung 3.4 aufgezeigten Ablauflogik. Dabei werden zunächst für alle Eingabedateien mittels der DSLRootFactory DSLRoot-Objekte erzeugt. Auf diesen werden dann alle ExecutionUnits des DSL-Tools ausgeführt.

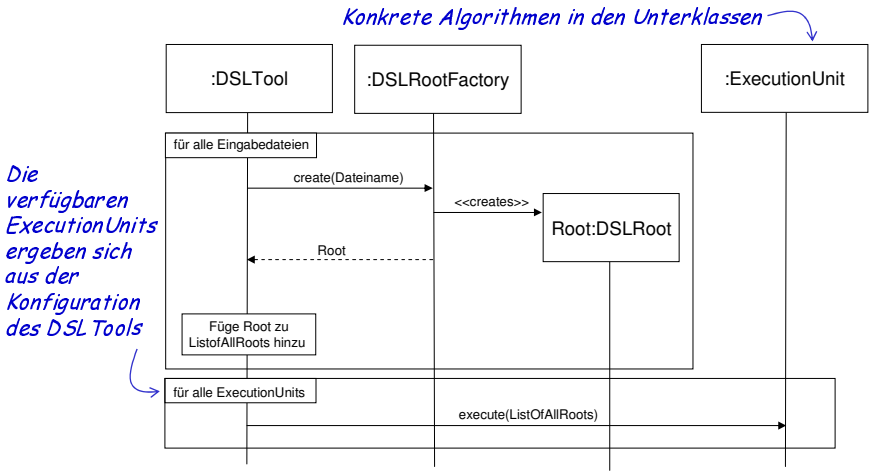


Abbildung 3.4.: Ablauf der Verarbeitung innerhalb eines DSLTools

Ein DSLTool wird typischerweise mit Hilfe eines weiteren Generators entwickelt. Dieses kann u.U. MontiCore selbst sein, aber auch andere DSLTool-Instanzen wie die MontiCore/UML[1] sind denkbar. Bei der Entwicklung wird die folgende Verzeichnisstruktur empfohlen, die in Abbildung 3.5 illustriert wird. Die einzelnen Pfade beziehen sich auf Unterordner des jeweiligen Projekts. Der verwendetete Codegenerator besitzt meistens eine Laufzeitbibliothek, die von den generierten Klassen verwendet wird. Diese wird als Softwarebibliothek eingebunden. Die Bezeichnung Laufzeitbibliothek kommt daher, dass diese Bibliothek zur Laufzeit der *entstehenden* Software läuft.

Das durch den Nutzer entwickelte DSLTool besteht zur Laufzeit aus verschiedenen Klassen, die direkt aus Java-Quellcode hervorgehen. Diese sind in Abbildung 3.6 mit Laufzeitdokument gekennzeichnet. Die Dokumente, die der Entwickler direkt beeinflussen kann, um die entstehende Software zu modifizieren, werden als Entwurfsdokumente bezeichnet. Mit dieser Klassifikation soll insbesondere herausgestellt werden, dass die generierten Klassen nicht modifiziert werden sollen und die Testfälle zur Qualitätssicherung beitragen, nicht aber das Laufzeitverhalten der Software beeinflussen.

Bei der Erstellung eines eigenen DSLTools wird der Entwickler durch MontiCore vor allem durch die Generierung von Parsern für die Eingabesprachen und eine standardisierte Verarbeitung unterstützt. Die Programmierung der Codegenerierung kann durch so genannte Template-Engines vereinfacht werden. MontiCore bietet hierfür eine eigene Engine an, die Refactorings besser unterstützt als herkömmliche Lösungen [KR05]. Ebenfalls nützlich für einfache Codegenerierungen ist das Besucher-Muster, das für die MontiCore-Klassen implementiert ist.

[1] Ein sich in der Entwicklung befindliches DSLTool, das die UML/P [Rum04b, Rum04a] zur Softwareentwicklung verfügbar macht.

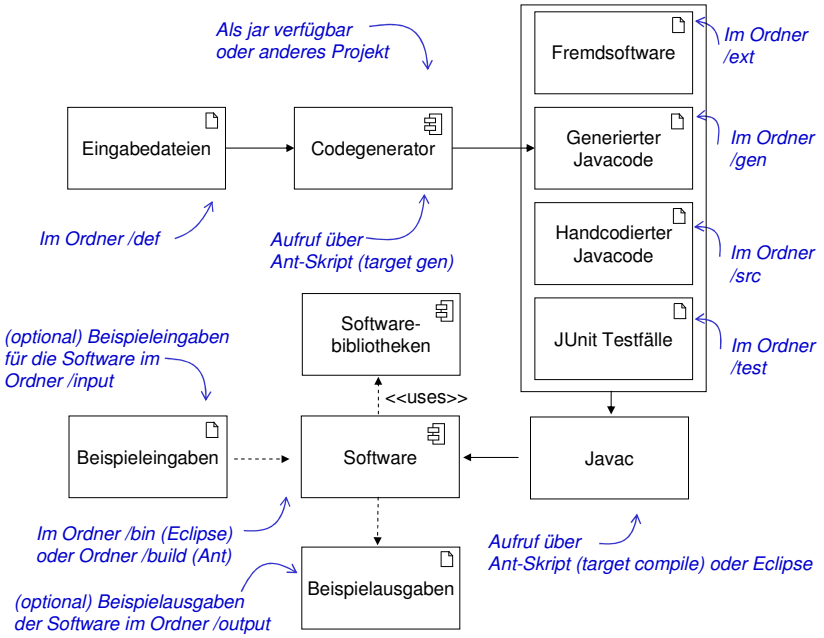


Abbildung 3.5.: Verzeichnisstruktur eines Codegenerators

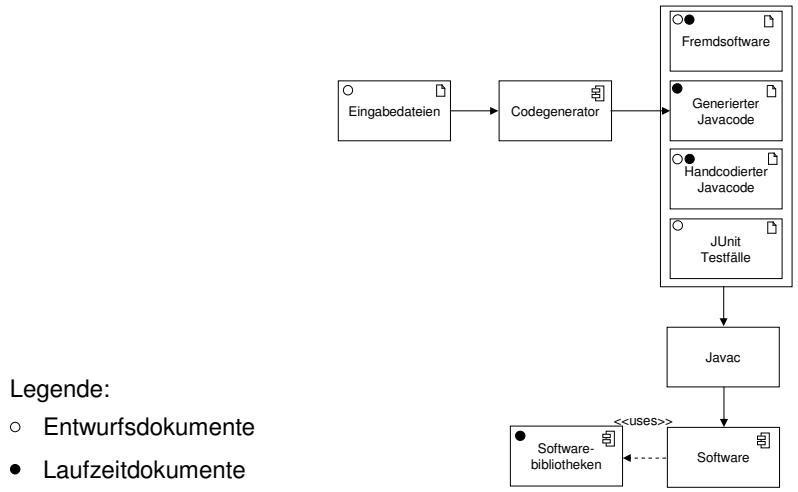


Abbildung 3.6.: Entwurfs- und Laufzeitdokumente

3.2. MontiCore

MontiCore ist ein Generator der selbst das MontiCore-DSLTool-Framework verwendet. Dadurch können drei verschiedene Eingabeformate verarbeiten werden:

MontiCore-Klassenbeschreibungssprache

Die MontiCore-Klassenbeschreibungssprache dient zur Spezifikation von Datenstrukturen im MontiCore-AST-Format. Das Dateiformat und die generierten Dateien werden ausführlich in Kapitel 4 erklärt.

MontiCore-Grammatikbeschreibungssprache

Das MontiCore-Grammatikformat ist die übliche Eingabe für einen Generierungsprozess. Das Dateiformat und die entstehenden Klassen werden ausführlich in Kapitel 5 erklärt.

Antlr

MontiCore kann aufgrund der Integration von Antlr 2.74 [WWWb] normale Antlr-Eingabedateien verarbeiten. Dazu ist lediglich ein zusätzlicher Kommentar in die erste Zeile einer Antlr-Grammatik aufzunehmen, der folgendes Format hat:
// antlr package "test.package"
Dabei steht „test.package“ stellvertretend für den Paketnamen der entstehenden Klassen. Antlr-Grammatikdateien haben die übliche Dateiendung „.g“.

Aufgrund der Verwendung von Antlr und der MontiCore-TemplateEngine [KR05] werden die Verzeichnisse entsprechend Abbildung 3.7 organisiert. Die Templates müssen als Quellen zur Laufzeit der entstehenden Software verfügbar sein, sind aber aufgrund der Besonderheiten der MontiCore-TemplateEngine ebenfalls im Projekt enthalten. Diese Konstruktion erlaubt teilweise die Refaktorisierung der Templates.

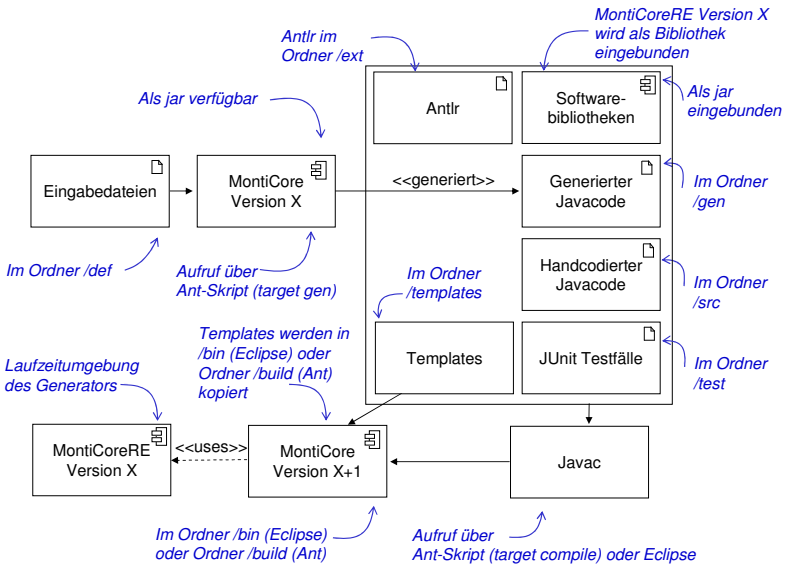


Abbildung 3.7.: Projektorganisation von MontiCore

Die MontiCore-Laufzeitbibliothek `de.monticore.re` wird zur Laufzeit der entstehenden Software verwendet und daher als Softwarebibliothek eingebunden. Dabei entspricht die Version der Laufzeitbibliothek stets der Version des verwendeten Generators. Somit läuft die MontiCore-Version X+1 mit der Laufzeitbibliothek Version X, da zur Erstellung die Version X des Generators verwendet wurde. Die Entwicklung der Laufzeitbiblitohek erfolgt bei MontiCore nicht streng abwärtskompatibel. Es wird jedoch darauf geachtet, dass sich nicht-kompatible Änderungen in der direkt darauf folgenden Version nicht auswirken. Daher kann die MontiCore-Version X+1 in der Regel auch die Laufzeitbibliothek X+1 verwenden.

3.3. Kopplung mehrerer Generatoren

Bisher wurde die Entwicklung und der Einsatz eines Codegenerators beschrieben, ohne jedoch auf die Funktion der entstehenden Software einzugehen. Diese Software kann wiederum ein Generator oder sogar ein Codegenerator sein, für den dieselben Prinzipien und Verzeichnisorganisationsregeln gelten, wie für das erste Generatorprojekt. Daraus ergibt sich ein im Prinzip beliebig tief schachtelbare Reihung von Codegeneratoren. Typischerweise werden jedoch nur zwei Generatoren benötigt wie in Abbildung 3.8 gezeigt wird.

Die Kopplung mehrerer solcher Werkzeuge ergibt sich aus folgendem Nutzungsszenario. Die Angaben in Klammern bezeichnen jeweils die Komponenten in Abbildung 3.8. Bei der Erstellung einer Software modifiziert der Entwickler typischerweise nicht nur den handcodierten Quellcode der Software (Handcodierter Javacode') und die Eingabedateien (Eingabedateien'), die die Informationen für den generierten Javacode (Generierter Javacode') enthalten. Vielmehr wird er auch das Verhalten des Codegenerator (Codegenerator') anpassen wollen, um spezifischen Quellcode für sein Projekt zu generieren. Dazu ist zum Beispiel eine Modifikation des handcodierten Quellcodes (Handcodierter Javacode) bzw. der Eingabedateien (Eingabedateien) notwendig.

Aus obiger Situation ergibt sich das Problem, dass in diesem zweischrittigen Prozeß für einen Entwickler nicht mehr offensichtlich ist, welche Teile seiner Software nach einer Modifikation neu übersetzt werden müssen. Insbesondere führt eine Veränderung der Quelldateien des ersten Generators (Codegenerator') dazu, dass alle Eingabedateien der zweiten Stufe (Eingabedateien') neu übersetzt werden müssen, da der Codegenerator (Codegenerator') sein Verhalten geändert haben kann. Der Einsatz von Automatisierungstechniken für den Buildprozess durch entsprechende Werkzeuge wie ant[WWWa] wird daher empfohlen und hat sich innerhalb der MontiCore-Entwicklung bewährt.

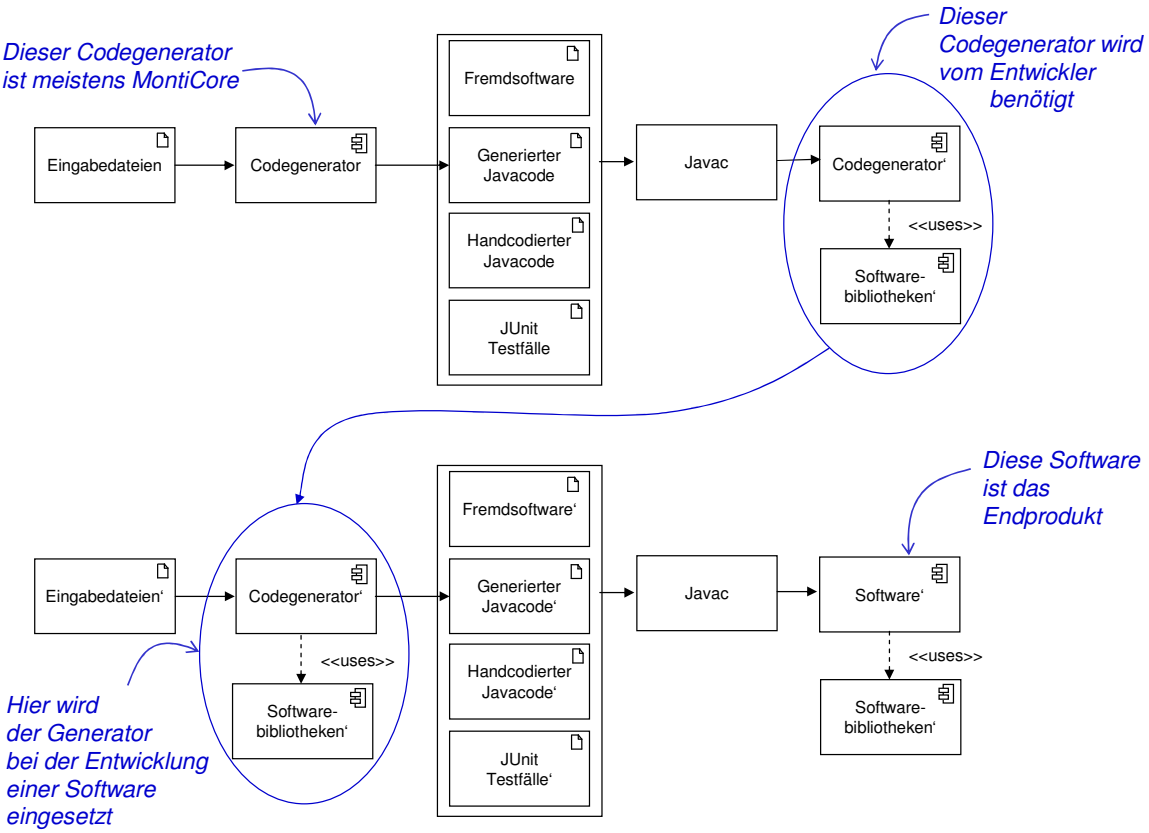


Abbildung 3.8.: Kopplung zweier Codegeneratoren

4. MontiCore-Klassenbeschreibungssprache

Ein wichtiger Bestandteil von MontiCore ist die MontiCore-Klassenbeschreibungssprache zur Beschreibung und Generierung von Klassenstrukturen für abstrakten Syntaxbäume (ASTs). Sie wird auf folgende Weisen verwendet:

1. Als eigenständige DSL zur Erzeugung beliebiger AST-Strukturen (die auch unabhängig von einer mit MontiCore erzeugten DSL einsetzbar sind) wie in Abb. 4.1(a) gezeigt.
2. Als MontiCore-interner Mechanismus zur Erzeugung einer AST-Struktur für eine Grammatikdefinition zu der von anderen MontiCore-Komponenten auch Lexer und Parser generiert werden, die diese AST-Struktur verwenden und aufbauen wie in Abb. 4.1(b).

Um eine möglichste flexible und sichere Verwendung der generierten Klassen zu erreichen, sind folgende Anforderungen bei der Umsetzung der MontiCore-Klassenbeschreibungssprache berücksichtigt:

- Die (objektorientierte) Zielsprache, in der die AST-Klassen erzeugt werden, ist prinzipiell austauschbar (obwohl die MontiCore-Klassenbeschreibungssprache zur Zeit auf Java-Codegenerierung ausgerichtet ist).
- Die AST-Knoten sind typisiert. Das führt dazu, dass nur strukturell korrekte ASTs erzeugt werden können und erhöht die Analysierbarkeit.
- Ein direkter Zugriff auf Kindknoten einer AST-Klasse ist über ein benanntes Attribut möglich.
- Eine Traversierung aller AST-Klassen ist durch ein Visitor-Muster realisiert.

Im Folgenden werden zunächst die generelle Struktur, besondere Eigenschaften und zusätzliche Hilfsfunktionen der erzeugten AST-Klassen dargestellt. Es folgt die Definition der Sprache und der Codegenerierung der MontiCore-Klassenbeschreibungssprache, die an einem Beispiel verdeutlicht werden. Das Beispiel zeigt die erste Verwendungsmöglichkeit, nämlich die Nutzung als eigenständige DSL. Abschließend wird die Integration in MontiCore, also die zweite Verwendungsmöglichkeit der MontiCore-Klassenbeschreibungssprache, diskutiert.

4.1. Die AST-Struktur

Im Folgenden wird zunächst der generelle Aufbau der MontiCore-AST-Klassen und deren Grundfunktionen beschrieben.

Alle mit der MontiCore-Klassenbeschreibungssprache erzeugten Klassen unterliegen einer bestimmten Struktur. Zentrales Element ist hierbei das Interface `ASTNode`, das von allen AST-Klassen implementiert wird, um das Vorhandensein wichtiger Basis-Methoden sicherzustellen. Hierzu gehören die Methoden `deepClone` zur rekursiven Kopie eines (Teil-)ASTs, `traverse` zur Traversierung des Knotens und seiner Unterknoten durch einen Visitor und Zugriffsmethoden für Vaterknoten (`parent`), zum Knoten gehörige, aus der Quelldatei stammende Kommentare (`preComment`, `postComment`) und Anfangs- und Endquellposition (`SourcePositionStart`, `SourcePositionEnd`). Die Zugriffsmethoden werden mit einem Unterstrich (nach „get“ bzw. „set“) versehen, um mögliche Kollisionen mit generierten Methoden zu vermeiden. Das Interface wird von AST-Klassen nicht direkt implementiert, sie erben normalerweise von der abstrakten Oberklasse `ASTCNode`, die für viele Methoden eine Standardimplementierung enthält. Ebenfalls können die AST-Klassen durch die Implementierung verschiedener Interfaces gemeinsamen Typen zugeordnet werden. Eine Besonderheit stellen Listen von AST-Klassen dar, sie werden durch Wrapperklassen realisiert und sind ebenfalls eigenständige AST-Klassen. Konkret erben Listenklassen von der abstrakten Klasse `ASTCList`, die zusätzlich zur Funktionalität der Klasse `ASTCNode` die Unterscheidung erlaubt, ob eine Liste leer oder nicht initialisiert ist.

Die Abbildung 4.2 zeigt einerseits die zur MontiCore-Laufzeitumgebung gehörenden Klassen im Paket `mc.ast`, andererseits ein Beispiel für AST-Klassen im Paket `mc.gen`, die gegen die Schnittstellen aus Paket `mc.ast` generiert werden. Die generierten AST-Klassen `A` und `B` erben wie beschrieben von `ASTCNode`. Sie implementieren darüber hinaus ein Interface `CommonType`. Listen von AST-Klassen werden durch Wrapperklassen wie im Beispiel `CommonTypeList` oder `BList` realisiert, erben von `ASTCList` und sind damit ebenfalls eigenständige AST-Klassen. Sie garantieren das Einfügen und Auslesen von AST-Klassen eines bestimmten Typs. Im Beispiel können sowohl Objekte vom Typ `A` oder `B` in die Liste `CommonTypeList` aufgenommen werden, wohingegen in der Liste `BList` nur Elemente vom Typ `B` erlaubt sind.

4.2. Die MontiCore-Klassenbeschreibungssprache

Ziel der Sprache ist es, Klassenstrukturen wie die in Abbildung 4.2 im Paket `mc.gen` einfach beschreiben und erzeugen zu können. Es ist wichtig zu betonen, dass mit Hilfe der MontiCore-Klassenbeschreibungssprache nicht die Abbildung von Grammatikregeln auf AST-Klassen beschrieben wird (siehe dazu Kapitel 5), sondern die Beschreibung von beliebigen AST-Strukturen.

4.2.1. Sprachdefinition und Codegenerierung

Die Definition der Sprache erfolgt über die Angabe ihrer Grammatik in MontiCore-Notation (Abbildung 4.3). Zugunsten einer kompakteren Darstellung werden einige technische Grammatikdetails ausgelassen. Die folgende Auflistung erläutert, wie mit Hilfe der MontiCore-Klassenbeschreibungssprache die Struktur eines AST definiert werden kann und was dies für die Codegenerierung bedeutet, bei der Java-Klassen generiert werden.

AstDescription (Zeile 7) Jede AST-Beschreibung beginnt mit dem Schlüsselwort `astdescription`. Hinter dieser Anweisung werden globale Einstellungen getroffen, die die Generierung aller Klassen beeinflussen.

DslName Der Name der DSL, zu der die AST-Klassen generiert werden. Dieser Name hat derzeit auf die Codegenerierung keinen Einfluss.

DefaultSuperClass Die Superklasse, von der im Normalfall alle generierten AST-Klassen erben. Dies ist für gewöhnlich die bereits vordefinierte Klasse `ASTCNode`.

DefaultSuperInterface Das Interface, das im Normalfall von allen zu erzeugenden AST-Klassen implementiert wird. Dies ist für gewöhnlich das beteits vordefinierte Interface `ASTNode`. Falls eine generierte Klasse bereits `ASTCNode` erweitert, wird diese Angabe ignoriert.

Visitor Die zu verwendende Visitor-Klasse für die zu generierenden `traverse`-Methoden. In MontiCore wird hier die Oberklasse `mc.ast.Visitor` verwendet.

DefaultPackage Das Paket, zu dem die zu generierenden Klassen gehören sollen.

OutputPath Das Verzeichnis, in das die Klassen geschrieben werden sollen.

File Eingeschlossen in geschweifte Klammern folgt die Definition beliebig vieler „Dateiinhalte“, dabei kann es sich konkret um Klassen- oder Interfacedefinitionen handeln. Bei der Codegenerierung wird entsprechend der Liste der angegeben Dateien für jedes Element eine Datei mit Java-Quellcode erzeugt.

Nach diesen Angaben erfolgt mit Hilfe der folgenden Anweisungen die Beschreibung der zu generierenden Klassen.

ClassDef (Zeile 18) Diese Regel leitet die Definition einer zu generierenden AST-Klasse ein. Nach Angabe der Klassennamens kann in Ausnahmefällen mit `noastclass` angegeben werden, dass es sich um keine AST-Klasse handelt. Dies verhindert die Generierung der `traverse`-Methode zur Unterstützung des

Visitor-Musters und der `deepClone`-Methode zum Replizieren eines vollständigen AST-Astes. Analog zur Java-Syntax können optional mit `extends` eine Superklasse und mit `implements` mehrere Interface-Klassen angegeben werden, die erweitert bzw. implementiert werden sollen. In den anschließenden geschweiften Klammern folgen die Details zur Beschreibung der Klasse mittels der Beschreibung von `AttribDef`, `Flag`, `FlagDef`, `Consts` und `Method`.

InterDef (Zeile 25) Neben Klassen sind auch Schnittstellen definier- und synthetisierbar. Wie bereits beschrieben, können sie beispielsweise zur Typisierung von Listenelementen eingesetzt werden. Die Definition eines Interface ähnelt der einer Klassendefinition. Alle Deklarationen, die mit den Befehlen `AttribDef`, `Flag`, `FlagDef`, `Const` und `Method` für ein Interface definiert wurden, werden allerdings nicht in die generierte Interface-Java-Datei, sondern statt dessen in alle Klassen geschrieben, die das Interface implementieren. Ein Interface kann analog zu Java optional mehrere Interfaces mit dem Schlüsselwort `extends` erweitern.

AttribDef (Zeile 31) Mit Hilfe von AttribDef lassen sich verschiedene Arten von Klassenattribute generieren, die im Folgenden anhand der unterschiedlichen Attributtypen `attribute`, `son`, `list` beschrieben werden.

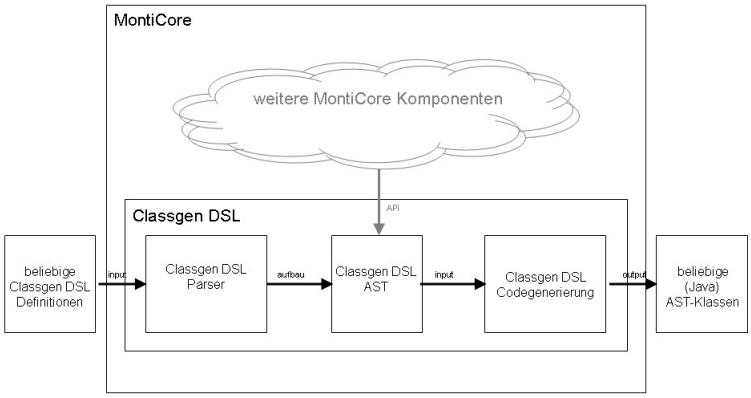
- **attribute** Generiert unter Angabe des Datentyps und des Variablennames ein Attribut einer Klasse mit entsprechenden Zugriffsmethoden. Diese Attribute bilden keine eingenständigen AST-Klassen und werden nicht in die `traverse`-Methode aufgenommen.
- **list** Erzeugt ein Klassenattribut für eine Liste von AST-Objekten eines angegebenen Typs mit entsprechenden Zugriffsmethoden. Dazu werden auch entsprechende Listenklassen generiert, die typsichere Operationen auf der Liste garantieren. Die Liste wird als Bestandteil des AST in die `traverse`-Methode integriert.
- **son** Erzeugt ein Klassenattribut für ein Objekt, das als Kind an die AST-Klasse angefügt werden kann, der bekanntlich einen Knoten im abstrakten Syntaxbaum darstellt. Diese Attribute werden in die `traverse` und `deepClone`-Methoden eingebunden.

Method (Zeile 36) Erlaubt die Erzeugung einer benutzerdefinierten Methode, die direkt in die generierte Klasse übernommen wird. Dabei können auch alle Methoden, die vom Klassengenerator erzeugt werden, von benutzerdefinierten Methoden ersetzt werden. Eine syntaktische Überprüfung der resultierenden Klasse findet allerdings nicht statt.

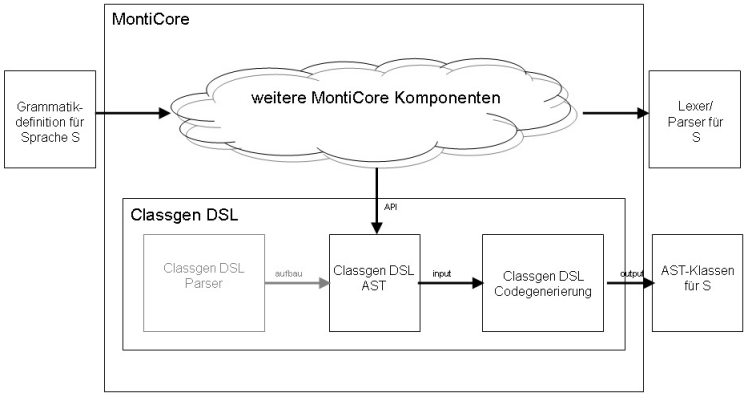
Consts (Zeile 35) Ermöglicht die Definition einer Integer-Konstanten in einer Klasse oder einem Interface. Der verwendete Integer-Wert wird fortlaufend inkrementiert.

FlagDef (Zeile 34) Definiert eine Gruppe von Flags. Dieser Befehl erzeugt Integer-Konstanten mit exponentiell aufsteigenden Werten, die daher miteinander in einem Integer kombinierbar sind.

Flag (Zeile 33) Generiert ein Integer-Attribut zur Verwendung der mit `flagdef` definierten Konstanten. Ergänzend werden entsprechende Zugriffsmethoden erzeugt.



(a) Für eine beliebige Instanz der MontiCore-Klassenbeschreibungssprache (Classgen DSL) werden entsprechende Java AST-Klassen generiert.



(b) Für eine Grammatikdefinition als Eingabe in MontiCore werden neben Parser und Lexer durch andere Komponenten auch die dazugehörigen AST-Klassen durch die MontiCore-Klassenbeschreibungssprache (Classgen DSL) erzeugt.

Abbildung 4.1.: Verwendungsmöglichkeiten der MontiCore-Klassenbeschreibungssprache. Die ausgegrauten Bereiche deuten an, welche Komponenten in der jeweiligen Verwendungsmöglichkeit nicht beteilig sind.

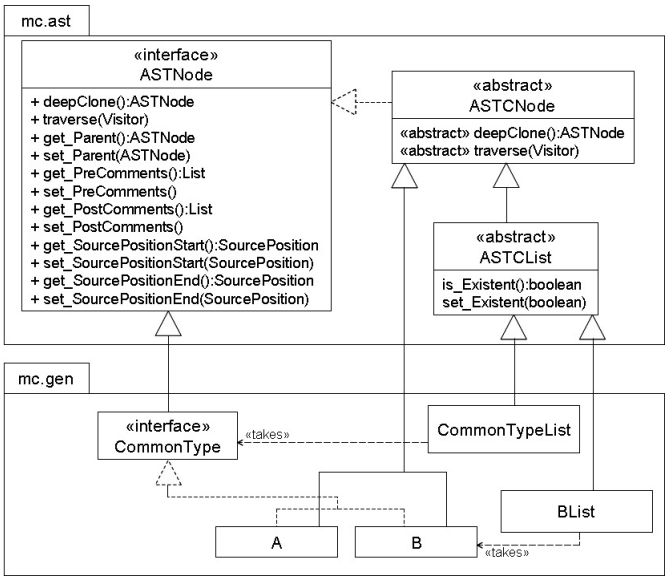


Abbildung 4.2.: Beispiel der AST-Klassenstruktur

```
1  package mc.classgen;
2
3  grammar ClassGen {
4
5    ident JAVA "'\\'''! (JESC|~('\\'''|'\\\\'))* '\\'''!";
6
7    AstDescription = !"astdescription"
8                     (!"dslname" Dslname:IDENT)
9                     (!"defaultsuperclass" DefaultSuperClass:IDENT)
10                    (!"defaultsuperinterface" DefaultSuperInterface:IDENT)
11                    (!"visitor" Visitor:IDENT)
12                    (!"defaultpackage" DefaultPackage:IDENT)
13                    (!"outputpath" OutputPath:STRING)
14                    "{" (Files:File)* "}";
15
16   Name =  Name:IDENT;
17
18   ClassDef(File)= !"class" ClassName:IDENT (Noastclass:[!"noastclass"])?
19                   (!"extends" Superclass:IDENT)?
20                   (!"implements" Interfaces:Name ("," Interfaces:Name)*)?
21                   "{" ( Attributes:AttribDef |
22                         FlagDefinitions:FlagDef | Constants:Consts |
23                         Flags:Flag | Methods:Method)* "}";
24
25   InterDef(File)= !"interface" InterfaceName:IDENT
26           (!"extends" Superinterfaces:Name ("," Superinterfaces:Name)*)?
27           "{" (        Attributes:AttribDef |
28           FlagDefinitions:FlagDef | Constants:Consts |
29           Flags:Flag | Methods:Method)* "}";
30
31   AttribDef= AttributeType:[!"attribute"|!"son"|!"list"]
32              ObjectType:IDENT Name:IDENT ";";
33   Flag= !"flag" GroupName:IDENT Name:IDENT ";";
34   FlagDef= !"flagdef" GroupName:IDENT "{" Flags:Name ("," Flags:Name)* "}" ";";
35   Consts= !"const" Constants:Name ("," Constants:Name)* ";";
36   Method= (Comment: JavaDocComment)? !"method"
37         (Public:[!"public"] | Private:[!"private"] | Protected:[!"protected"] |
38          Final:[!"final"] | Static:[!"static"])* (ReturnType:IDENT)?
39          Name:IDENT "(" Parameters:Parameter ("," Parameters:Parameter)* ")"
40         (!"throws" Exceptions:Name ("," Exceptions:Name)* )* Body:JAVA ";";
41
42   Parameter= Type:IDENT Name:IDENT;
43   JavaDocComment= !"comment" Comment:STRING;
44 }
```

Abbildung 4.3.: Grammatik der MontiCore-Klassenbeschreibungssprache

4.2.2. Klassengenerierung am Beispiel

```
1  astdescription
2    dslname classgenDemo
3    defaultsuperclass mc.ast.ASTCNode
4    defaultsuperinterface mc.ast.ASTNode
5    visitor mc.ast.Visitor
6    defaultpackage mc.classgen
7    outputpath "exampleout" {
8
9    class ASTMain {
10     son ASTInterface mySon;
11   }
12   interface ASTInterface {
13     attribute String name;
14     const A, B, C;
15     flagdef group1 {flag1, flag2};
16   }
17   class ASTSon implements ASTInterface {
18           list ASTElement elements;
19           method public String getAuthor() 'return "SSE";';
20   }
21   class ASTElement {
22     flag group1 flags;
23   }
24 }
```

Abbildung 4.4.: Beispieleingabe für die MontiCore-Klassenbeschreibungssprache

Zur Verdeutlichung zeigt dieser Abschnitt eine einfache Eingabe für die MontiCore-Klassenbeschreibungssprache (Abbildung 4.4) und einige der daraus generierten Java-Klassen, die sich dann im Verzeichnis `exampleout` (Zeile 7) und im Paket `mc.classgen` (Zeile 6) befinden.

Die erste definierte Klasse `ASTMain` (Zeile 9) enthält lediglich einen Kind-Knoten, welcher als Interface vorliegt. Bei der Codegenerierung wird daraus eine gleichnamige Java-Datei erzeugt (Abb. 4.5) und das Attribut für den Kind-Knoten, Konstruktor, die Zugriffsmethoden und die `deepClone` sowie die `traverse`-Methode generiert. Letztere bezieht das Attribut bei der Traversierung mit ein, da es Bestandteil des abstrakten Syntaxbaumes ist. Ebenfalls aus Abb. 4.5 zu erkennen ist der Effekt der Default-Oberklasse (Abb. 4.4, Zeile 3) und die Festlegung auf die Visitorklasse (Abb. 4.4, Zeile 5).

Die Interface-Definition `ASTInterface` (Zeile 12) enthält ein Attribut, drei Konstanten und eine Gruppe von Flags. Daraus ergibt sich der Code aus Abbildung 4.6. Hier greift auch Zeile 4 aus Abb. 4.4, das heißt, das Interface `ASTInterface` erweitert das Oberinterface `mc.ast.ASTNode`.

```
1  /* WARNING: This file has been generated, don't modify! */
2  package mc.classgen;
3  public class ASTMain extends mc.ast.ASTCNode {
4
5    protected ASTInterface mySon;
6
7    public ASTMain () {
8    }
9
10   public ASTMain (ASTInterface mySon) {
11      setMySon ( mySon );
12   }
13
14   public ASTInterface getMySon() {
15     return this.mySon;
16   }
17
18   public void setMySon(ASTInterface mySon) {
19     if (this.mySon != null) { this.mySon.set_Parent(null); }
20     this.mySon = mySon;
21     if (this.mySon != null) { this.mySon.set_Parent(this); }
22   }
23
24   public void traverse(mc.ast.Visitor visitor) {
25     visitor.visit(this);
26     visitor.startVisit(mySon);
27     visitor.endVisit(this);
28   }
29
30   public ASTMain deepClone(){
31     ASTMain result = new ASTMain();
32      if (mySon != null) {
33       result.setMySon((ASTInterface) mySon.deepClone());
34      }
35    /* ... */
36     return result;
37   }
38 }
39
```

Abbildung 4.5.: Generierter Quellcode aus der Klassendefinition `AstMain`

```
1 package mc.classgen;
2 public interface ASTInterface extends mc.ast.ASTNode {
3   public String getName();
4   public void setName(String name);
5   public static final int FLAG1 = 1;
6   public static final int FLAG2 = 2;
7   public static final int A = 0;
8   public static final int B = 1;
9   public static final int C = 2;
10 }
```

Abbildung 4.6.: Generierter Quellcode aus der Interfacedefinition `AstInterface`

Die Deklaration des im Interface definierten Attributs `name` erfolgt nicht in der Interfacedefinition selbst, sondern in allen das Interface implementierenden Klassen, hier Klasse `ASTSon` (Zeile 17). Der Quellcode findet sich zum Vergleich in Abbildung 4.7. Zu erkennen ist darin zudem die Umsetzung der `method`-Deklaration, die direkt in die Klasse übernommen wird. Des Weiteren beinhaltet die Definition von `ASTSon` eine Liste `elements` mit Objekten vom Typ `ASTElement`. Dies führt zur Generierung einer Listenklasse mit dem Namen `ASTElementList` Da die Liste Bestandteil des ASTs ist, wird diese im Gegensatz zum einfachen Attribut `name` in die `traverse`-Methode eingebunden.

Aus Platzgründen ist exemplarisch nur ein kleiner Ausschnitt der generierten Listenklasse `ASTElementList` in Abbildung 4.8 angegeben. Es ist erkennbar, dass die Klasse das Java-Collection Interface `List` implementiert und Aufrufe an die intern verwendete ArrayList delegiert werden. Für eine erhöhte Typsicherheit und einfacheren Zugriff sind die Listenklassen mit dem Typ parametrisiert, den sich aufnehmen können (hier zu sehen durch die Verwendung von Java 5.0 Generics mit Typparameter `ASTElement`).

Die generierte Klasse `ASTElement` nutzt schließlich in einer Variablen `myFlag` die im Interface definierten Flags. Demenstsprechend werden für die Klasse die zugehörigen Zugriffsmethoden, wie in Abbildung 4.9 dargestellt, erzeugt.

```
1  /* WARNING: This file has been generated, don't modify! */
2  package mc.classgen;
3  public class ASTSon extends mc.ast.ASTCNode implements ASTInterface {
4
5    public String getAuthor () {
6      return "SSE";
7    }
8
9    protected ASTElementList elements;
10
11   protected String name;
12
13   public ASTSon () {
14      setElements (new ASTElementList()) ;
15   }
16
17   public ASTElementList getElements() {
18     return this.elements;
19   }
20
21   public void setElements(ASTElementList elements) {
22     if (this.elements != null) { this.elements.set_Parent(null); }
23     this.elements = elements;
24     if (this.elements != null) { this.elements.set_Parent(this); }
25   }
26
27   public String getName() {
28     return this.name;
29   }
30
31   public void setName(String name) {
32     this.name = name;
33   }
34
35   public void traverse(mc.ast.Visitor visitor) {
36     visitor.visit(this);
37     visitor.startVisit(elements);
38     visitor.endVisit(this);
39   }
40
41   public ASTSon deepClone(){
42     ASTSon result = new ASTSon();
43      if (elements != null) {
44       result.setElements(elements.deepClone());
45      }
46     result.name = name;
47     /* ... */
48     return result;
49   }
50 }
51
```

Abbildung 4.7.: Generierter Quellcode aus der Klassendefinition `AstSon`

```
1  /* WARNING: This file has been generated, don't modify! */
2  package mc.classgen ;
3
4  import java.util.*;
5
6  import mc.ast.*;
7
8  public class ASTElementList extends mc.ast.ASTCList implements
9          java.lang.Iterable< ASTElement >, java.util.List< ASTElement > {
10
11   private ArrayList< ASTElement > list;
12
13   public ASTElement get(int index) {
14     return list.get(index);
15   }
16   /* ... */
17 }
```

Abbildung 4.8.: Generierter Quellcode aus der Listendefinition

```
1  public class ASTElement extends mc.ast.ASTCNode {
2
3    protected int flags = 0;
4
5    public ASTElement () {
6    }
7
8    public int getFlags() {
9      return this.flags;
10   }
11
12   public void setFlags(int flags) {
13     this.flags = flags;
14   }
15
16   public boolean isFlagsFlag1() {
17     return ((flags & ASTInterface.FLAG1) > 0);
18   }
19
20   public boolean isFlagsFlag2() {
21     return ((flags & ASTInterface.FLAG2) > 0);
22   }
23
24   public void setFlagsFlag1(boolean set) {
25     if (set) flags |= ASTInterface.FLAG1; else flags &= ~ASTInterface.FLAG1;
26   }
27
28   public void setFlagsFlag2(boolean set) {
29     if (set) flags |= ASTInterface.FLAG2; else flags &= ~ASTInterface.FLAG2;
30   }
31
32   /* ... */
33   }}
```

Abbildung 4.9.: Generierter Quellcode aus der Klassendefinition `ASTElement`

4.3. Integration der MontiCore-Klassenbeschreibungssprache in MontiCore

In diesem Abschnitt wird die zweite Verwendung der MontiCore-Klassenbeschreibungssprache gemäß Abbildung 4.1(b) beschrieben.

Nach dem Parsen einer beliebigen MontiCore-Grammatik müssen Lexer/Parser und AST-Klassen für diese Grammatik von MontiCore generiert werden. Um die gewünschte Klassenstruktur zu generieren, bestünde die Möglichkeit, zunächst die Struktur textuell mit Hilfe einer Eingabe für die MontiCore-Klassenbeschreibungssprache, wie sie Abbildung 4.4 zeigt, zu beschreiben und anschließend zu generieren. Bei diesem Schritt erfolgt intern natürlich zunächst das Parsen der Eingabe in den zur Grammatik (Abb. 4.3) gehörenden AST (siehe Abbilding 4.10), auf dem die Codegenerierungslogik arbeitet. Der Zwischenschritt über die textuelle Repräsentation kann übergangen werden und der AST direkt über die Programmierschnittstelle (API) aufgebaut werden, das heißt, Objekte des AST werden direkt in MontiCore instanziert und an die Codegenerierung übergeben (vgl. Abb. 4.1). In MontiCore wurde zur Generierung dieser Weg gewählt. Der spezielle Visitor `MTGrammar2Classgen` erledigt diese Aufgabe.

Erwähnenswert ist, dass die MontiCore-Klassenbeschreibungssprache selbst auf Basis von MontiCore entwickelt wird. Dabei tritt das Problem auf, dass Verarbeitung der Grammatikdefinition bereits die Existenz einer MontiCore-Klassenbeschreibungssprache bedingt. Folglich ist für die MontiCore-Klassenbeschreibungssprache ein teilweises bootstrapping in MontiCore erforderlich, das heißt, die Weiterentwicklung der MontiCore-Klassenbeschreibungssprache findet (nach einer initial handgeschriebenen Version) auf Basis ihrer generierten „Vorgängerversion“ statt (siehe Abbildung 4.11).

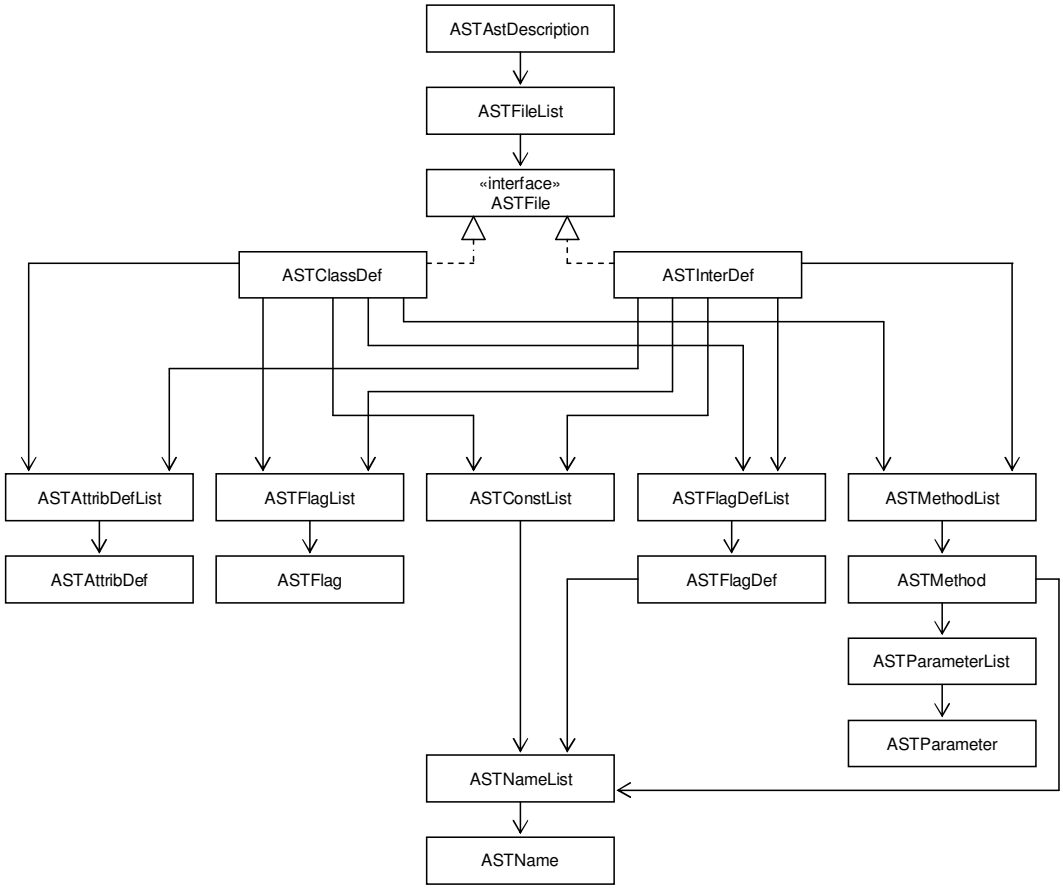


Abbildung 4.10.: AST des Klassengenerators

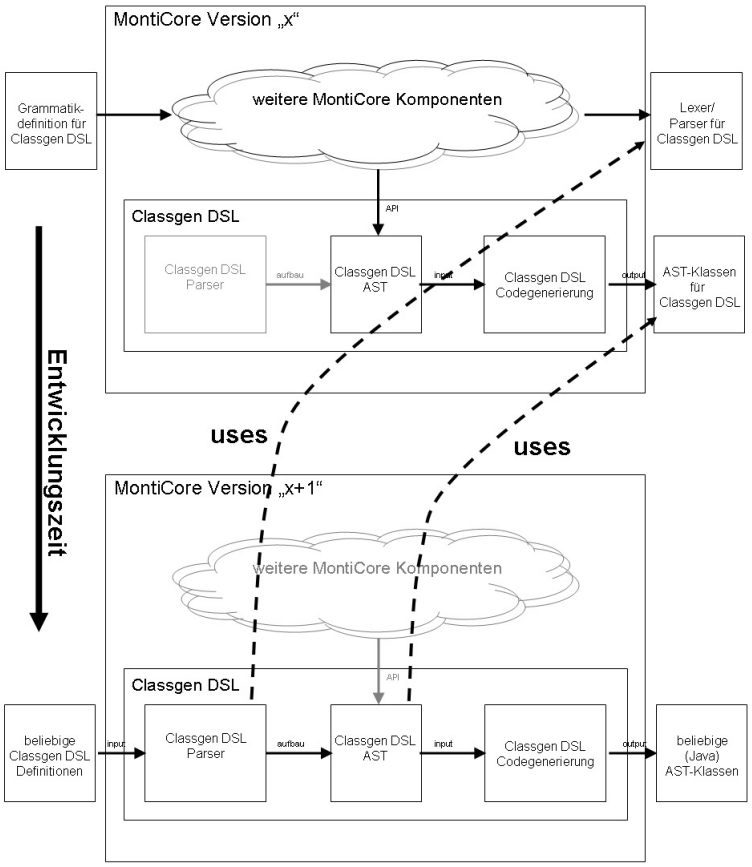


Abbildung 4.11.: Weiterentwicklung der MontiCore-Klassenbeschreibungssprache auf Basis ihrer Vorgängerversion

5. MontiCore Grammatik

Die MontiCore-Grammatikbeschreibungssprache dient zur Spezifikation einer Sprache in einer an die EBNF und Antlr-Notation angelehnten Form. Zusätzlich können Informationen und Hilfskonstrukte für eine einfache Verwendung der Sprache hinzugefügt werden. Das Grammatikformat ist aufgrund der unterliegender Parsertechnolgie (LL(k) erweitert um syntaktische und semantische Prädikate) in der Lage, gängige parsebare Sprachen zu beschreiben. Der Fokus von MontiCore liegt aber klar auf der Entwicklung von domänenspezifischen Sprachen, die meistens eine sehr einfache Syntax verwenden. Daher sind Komfort und Geschwindigkeit bei der Entwicklung eine Kernanforderung an MontiCore und weniger die Beschreibungsmächtigkeit der verwendeten Grammatiken.

Die Grundidee des Grammatikformats ist, dass diese Beschreibung ausreichend Informationen enthält, um AST-Klassen zu generieren und mit Hilfe von Antlr einen Parser zu erzeugen, der einen typisierten und heterogenen AST erzeugt. Die zugehörigen AST-Klassen unterscheiden sich von den AST-Klassen, die z.B. Antlr standardmäßig erzeugt, dadurch, dass sie den Direktzugriff auf Kinderknoten durch benannte Attribute erlauben (nicht über Konstrukte wie zum Beispiel ast.child[3]). Zusätzlich werden automatisch Listenklassen erzeugt und Klassen für den Zugriff auf den AST über ein Visitorenkonzept vorbereitet. Ergänzt wird die Definition der Sprache um so genannte Konzepte, die weitergehende Informationen an die Regeln annotieren und die Sprache über die reine syntaktische Beschreibung ergänzen.

Eine Grammatik besteht dabei aus einen Kopf und vier verschiedenen Teilen im Rumpf der Grammatik, die in beliebiger Reihenfolge aufgeführt werden können.

- Optionen, die Einstellungen für die Grammatik erlauben (siehe Abschnitt 5.1).
- Identifier, die einen Teil der lexikalischen Analyse bilden (siehe Abschnitt 5.2).
- Regeln, die die Grammatikregeln für die Spezifikation der Sprache darstellen (siehe Abschnitt 5.3).
- Konzepte, die die Möglichkeiten der Grammatik erweitern (siehe Abschnitt 5.4).

Das Gerüst einer Beispiel-Grammatik befindet sich in Abbildung 5.1. Außerdem befindet sich das Grammatikformat in einer EBNF-Beschreibung in Anhang B und im MontiCore-Format in Anhang C.

MontiCore-Grammatik

```
1 package example;
2
3 grammar ExampleGrammar {
4
5   // Optionen
6
7   // Identifier
8
9   // Regeln
10
11  // Konzepte
12 }
```

Abbildung 5.1.: Gerüst einer Beispiel-Grammatik im MontiCore-Format

5.1. Optionen

Die Codegenerierung aus einer MontiCore-Grammatik kann durch die Angabe von Optionen beeinflusst werden. Diese werden in einem Block angegeben, der mit dem Schlüsselwort `options` eingeleitet wird.

Durch die Angabe von `lexer` und `parser` kann zunächst der Lookahead festgelegt werden. Der Lookahead bezeichnet die Anzahl der Zeichen beim Lexer bzw. Wörter beim Parser, nach denen eine eindeutige Regelauswahl erfolgen kann. Fehlen diese Angaben, werden standardmäßig 3 Zeichen Lookahead für den Lexer und 1 Wort Lookahead für den Parser verwendet. Zusätzlich können weitere Optionen direkt an Antlr weitergereicht werden (vgl. http://www.antlr.org/doc/options.html), indem diese als Strings nach dem Lookahead eingefügt werden. Zusätzlich zu den Optionen für Lexer und Parser gibt es die Möglichkeit, in einem String nach dem Schlüsselwort `header` Quellcode für den Antlr-Kopf zu übergeben. Abbildung 5.2 zeigt beispielhaft einen Optionenblock.

MontiCore-Grammatik

```
1 options {
2   parser lookahead=1
3   lexer lookahead=3 "<<Antlr-Optionen>>"
4 }
```

Abbildung 5.2.: Optionen für eine MontiCore-Grammatik

Darüber hinaus können weitere Optionen in beliebiger Reihenfolge verwendet werden:

nows Standardmäßig werden Leerzeichen und Zeilenumbrüche während der lexikalischen Analyse ignoriert. Durch Setzen dieser Option können sie als Terminalzeichen in der Grammatik verwendet werden.

noslcomments Standardmäßig werden Zeichenketten, die mit // beginnen bis zum Zeilenende als Kommentar behandelt. Durch Setzen dieser Option wird dieses Verhalten deaktiviert und die Zeichen normal verarbeitet.

nomlcomments Standardmäßig werden Zeichen innerhalb der Begrenzungen /* und */ als Kommentar behandelt. Dieses Verhalten wird durch das Setzen dieser Option deaktiviert und die Zeichen normal verarbeitet.

noanything Standardmäßig wird ein Identifier MCANYTHING generiert, der in keiner Parserregel benutzt wird. Dieses Vorgehen ist für die Einbettung verschiedener Sprachen ineinander notwendig. Dieses Verhalten wird mit Setzen der Option deaktiviert.

noident Standardmäßig wird ein Identifier IDENT wie in Abschnitt 5.2 beschrieben generiert. Durch das Setzen der Option wird dieser Identifier nicht automatisch generiert, sondern kann durch den Nutzer angegeben werden.

nostring Standardmäßig wird ein Identifier STRING wie in Abschnitt 5.2 beschrieben generiert. Durch das Setzen der Option wird dieser Identifier nicht automatisch generiert, sondern kann durch den Nutzer angegeben werden.

nocharvocabulary Standardmäßig wird das Unicode-Eingabealphabet \u0003 bis \u7FFE für den Lexer verwendet. Dieses Verhalten wird mit Setzen der Option deaktiviert und kann dann in den Lexeroptionen in Antlr-Syntax ergänzt werden.

dotident Diese Option aktiviert einen erweiterten Identifier IDENT, der auch Punkte innerhalb eines Identifiers zulässt.

compilationunit X Diese Option sorgt für eine Einbindung der Grammatik in MontiCore-Framework und erzeugt zusätzliche Regeln für Paketinformationen und Imports innerhalb der Sprache. Das X verweist auf eine Regel der Grammatik, die ein Attribut Name vom Typ `java.lang.String` besitzt. Dieses ist z.B. der Fall, wenn eine Regelkomponente `Name:IDENT` verwendet wird. Diese Regeel wird nach dem Parsen von Paketnamen und Imports aufgerufen, bildet also die eigentliche Startregel der Grammatik.

5.2. Identifier

Das Parsen einer Eingabesequenz erfolgt, wie bei Compilern üblich, in einem zweistufigen Verfahren. Zunächst werden in der lexikalischen Analyse Wörter identifiziert,

die dann in der Syntaxanalyse zu einer hierarchischen Struktur kombiniert werden. MontiCore verwendet standardmäßig zwei Typen von Wörtern, IDENT und STRING, die durch die EBNF-Ausdrücke in Abbildung 5.3 definiert sind. Die Anwendung dieser Standards kann durch die entsprechenden Optionen unterbunden werden („noident“ und „nostring“, siehe Abschnitt 5.1).

EBNF

```
1 IDENT ::=
2   ( 'a'..'z' | 'A'..'Z' )
3   ( 'a'..'z' | 'A'..'Z' | '_' | '0'..'9' | '$' )*
4
5 STRING ::= " (<<Alle Zeichen außer Zeilenumbrüche>>)* "
```

Abbildung 5.3.: Übliche Identifier einer MontiCore-DSL

Unabhängig davon können eigene Wortdefinitionen ergänzt werden, wobei die in Abbildung 5.4 definierte Syntax verwendet werden muss. Dabei sollte als Konvention ein Name verwendet werden, der ausschließlich aus Großbuchstaben besteht. Danach können Optionen als String angegeben werden, die wie die nachfolgende Definition unverändert Antlr zur Verarbeitung übergeben werden. Typischerweise sollte „options testLiterals=true;“ verwendet werden, falls die Definition auch für Schlüsselwörter der domänenspezifischen Sprache zutrifft, um eine fehlerfreie Erkennung von Schlüsselwörtern zu gewährleisten. Schließlich folgt die eigentliche Definition als String. Durch eine Schrägstrich kann eine Regel als geschützt gekennzeichnet werden, wodurch beim Lexing diese Regel nicht direkt erfüllt wird, sondern nur als Unterregel von anderen Regeln aufgerufen werden kann.

MontiCore-Grammatik

```
1 LexRule =
2    !"ident" (Protected:[Protected:"/"])? Name:IDENT
3    (Option:STRING)? Symbol:STRING ";";
```

Abbildung 5.4.: MontiCore-Grammatikbeschreibung der Identifier-Definition

Bei der Definition eines Identifiers handelt es sich nicht um eine MontiCore-eigene Syntax, da diese Eingabedaten unverändert an das zur Parsererzeugung verwendete Antlr weitergereicht werden. Diese Daten werden von MontiCore als Strings geparst, was wie üblich dazu führt, dass z.B. ein Backslash als „\\“ geschrieben wird. Da Antlr dasselbe Verfahren in Bezug auf die Übersetzung in den in Java implementierten Parser verwendet, muss das Verfahren sogar doppelt angewendet werden: „\\\\“. Soll also ein Zeilenumbruch im regulären Ausdruck für Antlr („\n“) erscheinen, muss dieser als „\\n“ verwendet werden. Dieses Verfahren ist sicherlich nicht das komfortabelste und daher eine Verbesserungsmöglichkeit für MontiCore in naher

Zunkunft. Zwei Beispiele für Identifier-Definitionen in einer MontiCore-Grammatik befinden sich in Abbildung 5.5.

MontiCore-Grammatik

```
1 // standard identifier which may start with letters, a $ or an underscore
2 ident DOTIDENT
3         "options {testLiterals=true;}"
4         "( 'a'..'z' | 'A'..'Z' | '_' | '0'..'9' | '$' | '.')*";
5
6 ident NUMBER
7         "( '1'..'9' )+ ( '.' ('0'..'9')+ )?";
```

Abbildung 5.5.: Beispiele für eigene Identifier in einer MontiCore-DSL

Der Nachteil der umständlichen Definitionsweise wird dadurch ausgeglichen, dass sich die meisten von MontiCore adressierten DSLs mit den Standard-Identifiern `IDENT` und `STRING` darstellen lassen.

5.3. Regeln

Die Regeln einer MontiCore-Grammatik bestimmen zum einen den entstehenden Parser als auch die Struktur der AST-Klassen, die vom Parser instanziert werden. Im Folgenden wird die Übersetzung einer MontiCore-Grammatik dadurch illustriert, dass äquivalente EBNF-Produktionen und Klassendiagramme der AST-Klassen gezeigt werden. Die Auswahl beschränkt sich auf - wenn auch repräsentative - Beispiele. Eigene Experimente lassen sich am besten mit dem MontiCore-Compiler selbst ausführen: Die Datei «DSLName».ebnf zeigt immer die EBNF Darstellung der Grammatik, die mit der weitaus technischeren Darstellung in Form der Antlr-Grammatik («DSLName».g) übereinstimmt. Die entstehenden AST-Klassen werden von MontiCore im Quellcode erzeugt und können daher einfach selbst studiert werden. Alternativ wird ein Klassendiagramm als Postscript-Datei erzeugt, sofern GraphViz [WWWn] auf dem System verfügbar ist.

Grundsätzlich gilt für die AST-Klassengenerierung in MontiCore, dass aus jeder Regel eine AST-Klasse erzeugt wird. Abbildung 5.6 illustriert diese Generierung anhand einer einfachen Grammatikregel. Dabei werden aus dem rechten Teil einer Regel Attribute der AST-Klasse erzeugt.

Die Regelkomponenten, also die Elemente, die sich auf der rechten Seite einer Produktion befinden, können sowohl Terminale als auch Nichtterminale sein. Terminale bezeichnen Elemente der Grammatik, die atomar sind, wohingegen Nichtterminale auf andere Elemente der Grammatik verweisen und so weiter zerlegt werden können.

MontiCore-Grammar

```
1 A = Name:B ;
```

EBNF

```
1 A ::= B
```

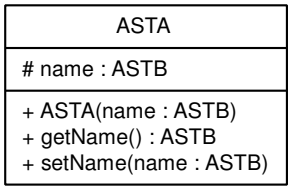


Abbildung 5.6.: Codegenerierung aus einer einfachen Produktion

Terminale und Nichtterminale in einer MontiCore-Grammatik haben einen ähnlichen Aufbau. In Abbildung 5.6 befindet sich eine einzelne Regelkomponente `Name:B`. Der vereinfachte Aufbau einer Regelkomponente lässt sich aus Abbildung 5.7 erkennen, wobei der erste Teil u.a. bestimmt, wie die Regelkomponente in den AST abgebildet wird. Dazu gibt es die folgenden drei Möglichkeiten:

ohne Bezeichner
: Der AST wird aufgebaut, ohne dass dieses Nichtterminal in den AST aufgenommen wird.

Bezeichner '='
: Der AST wird aufgebaut, ohne dass dieses Nichtterminal in den AST aufgenommen wird. Das Nichtterminal wird jedoch als Variable zur Verfügung gestellt.

Bezeichner ':'
: Das Nichtterminal wird als Attribut zur jeweiligen Klasse hinzugefügt.

EBNF

```
1 Regelkomponente ::=
2   ( IDENT ':' | IDENT '=')? X;
```

Abbildung 5.7.: Vereinfachte Darstellung einer Regelkomponente

Der zweite Teil einer Regelkomponente, in Abbildung 5.7 mit X bezeichnet, bestimmt den Typ des Attributs im AST bzw. der Variablen. Bei Nichtterminalen verweist das X entweder auf einen Identifier oder auf eine andere Regel. Bei Terminalen stehen an der Stelle X Zeichenketten, Schlüsselwörter, Konstanten oder Konstantengruppen, die in Abschnitt 5.3.2 näher erläutert werden.

5.3.1. Nichtterminale

Nichtterminale stellen eine Möglichkeit dar, eine Grammatik und damit auch die entstehenden AST-Klassen zu strukturieren. Nichtermininale verweisen dabei auf andere Regeln oder Identifier der Grammatik.

Die bereits beschriebenen Identifier lassen sich auf die in Abbildung 5.8 dargestellte Art und Weise verwenden, wobei zu beachten ist, dass diese unabhängig von ihrer Definition auf `java.lang.String` in den AST-Klassen abgebildet werden. Direktes Parsen von Zahlen lässt sich daher nur über Umwege gestalten. Auch hier zeichnet sich eine Erweiterungsmöglichkeit für MontiCore ab.

```
MontiCore-Grammar
1 A = Name:IDENT Value:STRING ;

EBNF
1 A ::= IDENT STRING

Beispieleingabe
1 Name "Wert"
```

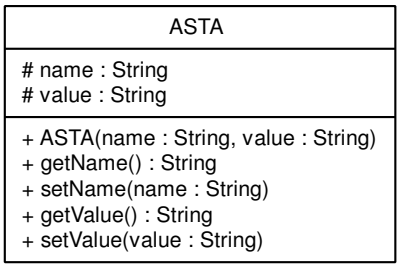


Abbildung 5.8.: Codegenerierung aus Identifiern

5.3.2. Terminale

Eine weitere Möglichkeit neben der Verwendung von Identifiern, Daten in AST-Klassen einzulesen, ist die Verwendung von Terminalsymbolen.

Werden Terminalsymbole in einer Grammatik verwendet, sehen sie den Nichtterminalen sehr ähnlich. Der erste Teil von Terminalen folgt dem bereits erläuterten Schema, so dass sich dieselbe Syntax wie in Abbildung 5.7 ergibt.

Für den zweiten Teil erlaubt MontiCore die Verwendung der folgenden Terminalsymbole.

Zeichenketten, wie z.B. “+“
: Zeichenketten und Schlüsselwörter dienen dazu, konstante Ausdrücke in eine Grammatik einzufügen. Die Abbildung konstanter Zeichenketten erfolgt auf `java.lang.String`.

Schlüsselwörter, wie z.B. !“public“
: MontiCore verwendet wie Antlr ein zweischrittiges Verfahren zum Erkennen von Schlüsselwörtern. Diese werden zunächst vom Lexer als Identifier erkannt, und dann in einem zweiten Schritt in ihrer Klassifikation verändert. Dieses Verfahren reduziert die Komplexität des Lexer und führt im Allgemeinen zu einem verbesserten Laufzeitverhalten. Die korrekte Funktionalität kann aber nur sichergestellt werden, wenn die Schlüsselwörter zunächst als Identifier erkannt werden. Anderfalls müssen diese - wie oben beschrieben - als Zeichenketten gekennzeichnet werden.

  Bis zu einer Implementierung der automatischen Erkennung durch MontiCore muss der Nutzer die beiden Fälle unterschieden: Zeichenketten, die von einer Identifier-Regel erfasst werden, und Zeichenketten, die zu keiner Identifier-Regel passen. Erstere müssen in MontiCore-Grammatiken durch ein Ausrufezeichen gekennzeichnet werden. Beide Fälle werden jedoch wie Zeichenketten behandelt und daher auf `java.lang.String` abgebildet.

Konstanten, wie z.B. [“+“]
: Konstanten unterscheiden sich konzeptuell von Identifiern dadurch, dass nicht Wertebereiche, sondern konkrete Zeichenketten vorgeben werden. In der jeweiligen AST-Klasse entstehen dann `booleans`, die aussagen, ob der jeweilige Wert angetroffen wurde oder nicht.

Konstantengruppen, wie z.B. [“-“,“+“]
: Für Konstantengruppen entsteht in den AST-Klassen ein `int`-Wert. Die konstanten Werte, die verwendet werden sollten, sind in einer Datei ASTConstantsDSLName abgelegt, wobei DSLName für den Namen der gerade entwickelten Sprache steht (siehe Abbildung 5.10). Wird eine Konstante bei der Nutzung der Sprache verwendet, wird die Integer-Variable auf den entsprechenden Wert der Konstante gesetzt, ansonsten auf den Wert von `default`.

Die Umsetzung von Konstanten lässt sich aus der Abbildung 5.9 erkennen. In Abbildung 5.10 wird gezeigt, dass vor einer Konstante ein zusätzlicher Namen angegeben werden kann, um die Bezeichnung der Konstante in der Konstantendatei zu beeinflussen. Abbildung 5.11 zeigt die typische Verwendung von Schlüsselwörtern, die meistens nicht in den AST abgebildet werden müssen.

5.3.3. Alternativen und Klammerstrukturen

Die bisher aufgezeigten Beispiele waren sehr einfach und verwenden keine Alternativen und Klammerstrukturen in Grammatikregeln. Diese sind jedoch in Grammatiken üblich und können auch im MontiCore-Grammatikformat verwendet werden.

MontiCore-Grammar

```
1 A = Modifier:[!"private"]
2     Protection:["/"] ;
```

EBNF

```
1 A ::= 'private' '/'
```

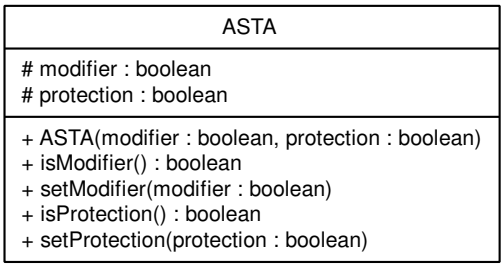


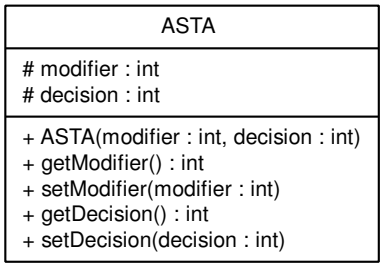
Abbildung 5.9.: Konstanten mit einem einzelnen Wert

MontiCore-Grammar

```
1 A = Modifier:[!"private",!"public"]
2     (Decision:[Yes:"+",No:"-"])? ;
```

EBNF

```
1 A ::= ('private'|'public')
2       ('+'|'-')?
```



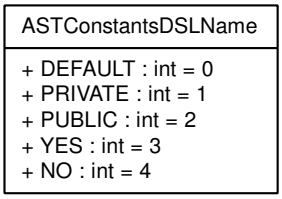


Abbildung 5.10.: Konstanten mit mehreren Wertmöglichkeiten

MontiCore-Grammar

```
1 A = !"keyword" ";" ;
```

EBNF

```
1 A ::= 'keyword' ';'
```

Abbildung 5.11.: Schlüsselwörter und Trennzeichen

MontiCore nutzt dieselbe Schreibweise wie EBNF und die Konzepte können unverändert auf die EBNF-Grammatik übertragen werden. Ein Fragezeichen am Ende eines Klammerblocks bedeutet, dass dieser optional ist. Ein */+ hinter Klammerblöcken sagt aus, dass der Block beliebig oft bzw. mindestens einmal wiederholt werden kann.

Die AST-Klassenerzeugung ergibt sich aus der Vereinigungsmenge aller Regelkomponenten und ist daher unabhängig von der Blockstruktur einer Regel. Somit entsteht

für die Beispiele aus Abbildung 5.12 stets dieselbe AST-Klasse.

```
MontiCore-Grammar
1 A = Name:B Value:C ;

EBNF
1 A ::= B C
```

```
MontiCore-Grammar
1 A = (Name:B | Value:C) ;

EBNF
1 A ::= (B | C)
```

```
MontiCore-Grammar
1 A = (Name:B | Value:C)? ;

EBNF
1 A ::= (B | C)?
```

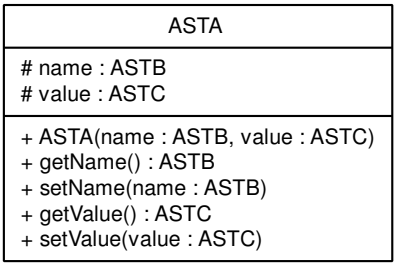


Abbildung 5.12.: Alternativen in der Grammatik

Bei der Verwendung von Klammerstrukturen mit * oder + ist die AST-Klassengenerierung komplexer als bei einfachen Attributen. Wie bereits erwähnt, entsteht die AST-Klasse durch die Vereinigungsmenge aller Regelkomponenten. Dabei ist die Reihenfolge der Regelkomponenten irrelevant.

Bei einem Nichtterminal innerhalb eines Blocks mit * oder + wird eine Listenklasse generiert, die nur Elemente diesen Typs aufnehmen kann. Dabei ist wiederum irrelevant, ob die Attribute Teil einer Alternative sind oder nicht, da der Grundsatz gilt, dass die Vereingungsmenge aller Regelkomponenten gebildet wird. Aus Abbildung 5.13 ist die Generierung der AST-Klassen ersichtlich.

Generell gilt in MontiCore, dass Namen innerhalb einer Produktion mehrfach verwendet werden dürfen. In Abbildung 5.14 lässt sich ein gutes Beispiel dafür sehen. Wichtig bei einer solchen Grammatikdefintion ist, dass die Reihenfolge von `name` und `value` bedeutungslos ist und daher auch nicht in der AST-Klasse abgebildet werden muss. Dieses muss der Nutzer sicherstellen, oder eine alternative Schreibweise wählen, wie sie später im Text und in Abbildung 5.18 beschrieben wird.

Ebenfalls möglich, aber weit weniger sinnvoll ist das Beispiel aus Abbildung 5.15. Es werden in diesem Fall zwei Identifier geparset und nacheinander dem Attribut name zugewiesen. Dabei enthält nach dem Parsen die AST-Klasse den Wert des zweiten Identifiers und der erste ist quasi verloren gegegangen. Durch den Parser wird dieses

```
MontiCore-Grammar
1 A = (Name:B)* (Value:C)* ;

EBNF
1 A ::= (B)* (C)*

MontiCore-Grammar
1 A = (Name:B | Value:C)* ;

EBNF
1 A ::= (B | C)*

MontiCore-Grammar
1 A = (Name:B | Value:C)+ ;

EBNF
1 A ::= (B | C)+
```

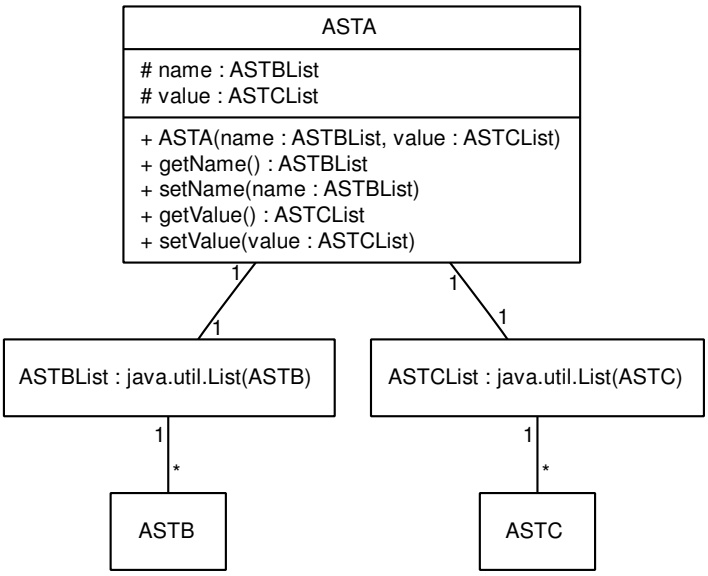


Abbildung 5.13.: Generierung von Listenstrukturen

```
MontiCore-Grammar
1 A = ( Name:B  Value:C
2     | Value:C Name:B  ) ";" ;

EBNF
1 A ::= (B C | C B)
```

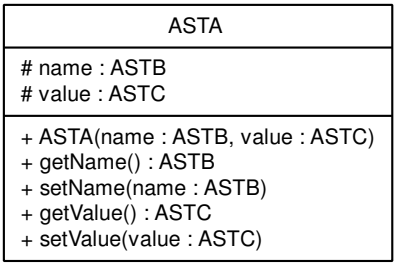


Abbildung 5.14.: Generierung von Attributen bei mehrfachem Auftreten

Verhalten mit einer Fehlermeldung quitiert. Falls ein solche Fehlermeldung unerwünscht ist, kann das Konzept `Classgen` verwendet werden (siehe Abschnitt 5.4.2). Falls dennoch eine feste Anzahl an Identifiern in einer Regel enthalten sein sollen, ohne dass diese überschrieben werden, müssen für diese verschiedene Namen vergeben werden.

Bei der Verwendung von Nichtterminalen gilt, dass aus Verweisen Attribute bzw. Listen entstehen, wenn diese in einem Block mit * oder + stehen. Bei Mischformen wie zum Beispiel in Abbildung 5.16 gezeigt, wird stets eine Liste generiert, in die alle Vorkommen dieses Verweises eingetragen werden.

Grundsätzlich analysiert MontiCore den Typ jedes Attributs und jeder Variablen

```
MontiCore-Grammar
1 A = Name:IDENT Name:IDENT ";" ;

EBNF
1 A ::= IDENT IDENT
```

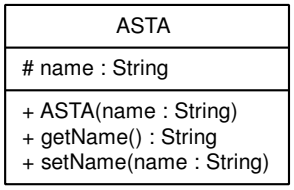


Abbildung 5.15.: Generierung von Attributen bei mehrfachem Auftreten

```
MontiCore-Grammar
1 A = Name:B ("," Name:B)* ;

EBNF
1 A ::= B ("," B)*
```

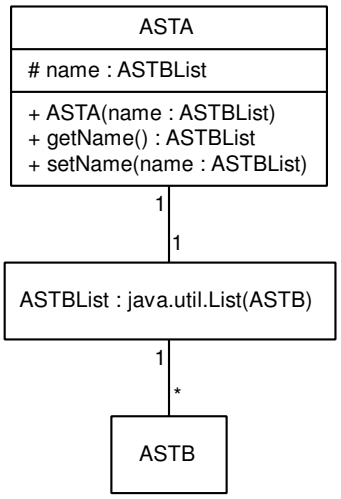


Abbildung 5.16.: Generierung von durch Trennzeichen separierte Listenstrukturen

innerhalb einer Regel. Dabei wird zunächst der Grundtyp bestimmt, wie z.B. die AST-Klasse für eine andere Regel, `java.lang.String` bei Identifiern und Zeichenketten, `boolean` bei Konstanten und `int` bei Konstantengruppen.

Ein Symbol, das als Attribut verwendet wird, wird als iteriert bezeichnet, wenn es innerhalb eines Blocks mit * oder + auftritt. Dieses kann auch direkt der Fall sein, da Blöcke beliebig geschachtelt werden dürfen. Ein Symbol, das als Variable verwendet wird, ist hingegen nur dann iteriert, wenn es selbst mit einem + oder * gekennzeichnet ist. Die Umgebung bleibt dabei unbeachtet. Das Standardverhalten ergibt sich aus der Tabelle 5.1, kann jedoch durch das Konzept `Classgen` (vgl. Abschnitt 5.4.2) beeinflusst werden.

Typ	nicht iteriert	iteriert	beides
Andere Regel	AST«Regelname»	AST«Regelname»List	AST«Regelname»List
Identifier/Zeichenkette	String	ASTStringList	ASTStringList
Konstante	boolean	boolean	boolean
Konstantengruppe	int	int	int

Tabelle 5.1.: MontiCore-Standardverhalten bei der Ableitung von Typen

5.3.4. Schnittstellenproduktionen

Im Folgenden werden zwei Möglichkeiten vorgestellt, eine komplexere Sprache durch eine Grammatik zu beschreiben. Die entstehenden AST-Klassen unterscheiden sich, so dass hier gut illustriert werden kann, wie Entwurfsentscheidung die Sprachdefinition beeinflussen. Solche Entscheidungen können durch den MontiCore-Nutzer selbst getroffen werden, um die enstehende Sprache optimal nutzen zu können.

Die Grundlage des folgenden Beispiels bildet eine rudimentäre Statechart-Grammatik, die Zustände und Transitionen enthält. In der Grammatik ist es möglich, innerhalb eines Zustandes Unterzustände und Transitionen anzugeben.

Falls die Reihenfolge von Unterzuständen und Transitionen keine Rolle spielt, kann es für die Benutzung des ASTs praktischer sein, zwei getrennte Listen für Transitionen und Unterzustände zu bilden, wie es aus Abbildung 5.17 zu erkennen ist. Die Reihenfolge der Transitionen und Unterzustände bleibt erhalten, die Reihenfolge zwischen den beiden Listen ist jedoch nicht festgehalten.

MontiCore-Grammar

```
1 State =
2   !"state" name:IDENT "{"
3     ( Substates:State
4     | Transitions:Transition )*
5   "}" ;
6 Transition = from:IDENT "->" to:IDENT
```

EBNF

```
1 State ::= 'state' IDENT '{'
2           (State|Transition)*
3           '}'
4 Transition ::= IDENT '->' IDENT
```

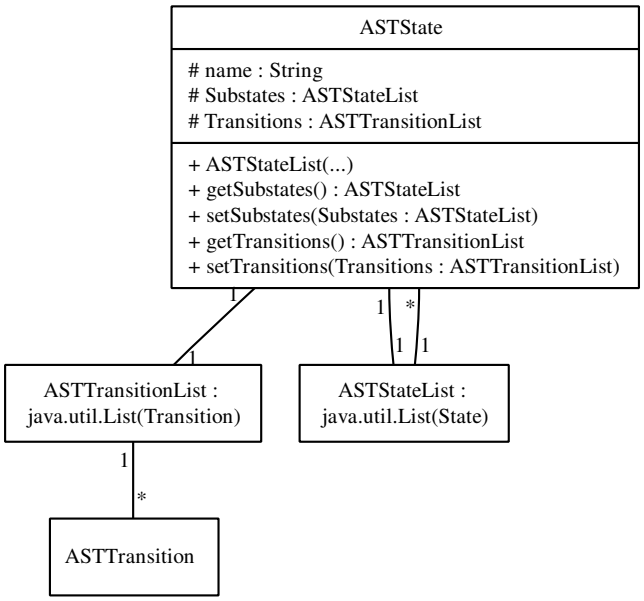


Abbildung 5.17.: Getrennte Listen für Transitionen und Unterzustände

Falls die Reihenfolge der Zustände und Transitionen untereinander für die Bedeutung der Sprache wichtig ist, kann die alternative Lösung aus Abbildung 5.18 verwendet werden. Dabei entsteht eine Liste für Transitionen und Unterzustände zugleich. Die Regeln `Transition` und `State` zeigen innerhalb der Klammern an, dass sie überall dort stehen können, wo ein `StateElement` stehen kann. Diese Notation erinnert an Oberklassenbildung in objekt-orientierten Programmiersprachen und spiegelt sich in der Erzeugung der AST-Klassen durch eine gemeinsame Oberschnittstelle wider. In einer Monticore-Grammatik ist es möglich, mehrere Schnittstellen durch Komma getrennt anzugeben.

MontiCore-Grammar

```
1 State (StateElement) =
2   !"state" name:IDENT "{"
3     (StateElements:StateElement)*
4   "}" ;
5
6 Transition (StateElement) =
7   from:IDENT "->" to:IDENT
```

EBNF

```
1 State ::= 'state' IDENT '{'
2           (StateElement)*
3           '}'
4 Transition ::= IDENT -> IDENT
5 StateElement ::= State | Transition
```

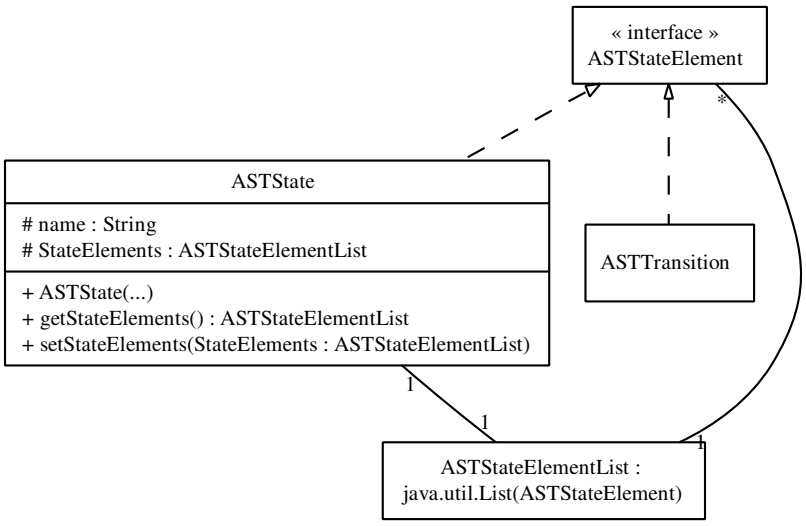


Abbildung 5.18.: Gemeinsame Liste für Transitionen und Unterzustände

5.3.5. Nutzung von Konstrukten aus Antlr

Neben den bereits beschriebenen Grundformen können weitergehende Konstrukte in einer Grammatik verwendet werden. Diese sind im Folgenden dokumentiert, verwenden jedoch zum Großteil die Antlr-Syntax und werden meistens auch unverändert an den unterliegenden Parser-Generator weitergereicht. Für eine detaillierte Beschrei-

bung und eine weitergehende Motivation, welche Konstrukte benötigt werden, siehe http://www.antlr.org/doc/.

Durch die Aufnahme dieser Konstrukte soll verhindert werden, dass es nötig wird, den generierten Parser zu verändern oder auf Antlr zurückgreifen zu müssen, weil sich eine spezielle Produktion in MontiCore-Syntax nicht beschreiben lässt. Weitergehende Möglichkeiten bietet das Konzept `Antlr`, das die Einbettung von Antlr- oder Java-Quellcode erlaubt (vgl. Abschnitt 5.4.4).

Syntaktische Prädikate

Die in MontiCore verwendete Parsetechnologie arbeitet standardmäßig mit einem festen Lookahead, d.h. einer konstanten Anzahl an Wörtern, die der Parser von der aktuellen Position nach vorn schaut, um Alternativen auszuwählen.

Betrachtet man die Grammatik in Abbildung 5.19 ohne das syntaktische Prädikat `(("a")* "b")=>` werden z.B. für einen Lookahead von zwei, Eingabestrings wie `ab` und `ac` erfolgreich geparst, der Eingabestring `aac` führt aber zu einem Fehler. Hier betrachtet Antlr nur die ersten zwei Eingabezeichen (`aa`) und kann daraufhin nicht die beiden Alternativen `A` oder `B` unterscheiden.

MontiCore-Grammatik

```
1 S = (("a")* "b")=> A
2     | B
3 A = ("a")* "b" ;
4 B = ("a")* "c" ;
```

Abbildung 5.19.: Syntaktische Prädikate für mehrdeutige Grammatiken (in Bezug auf einen festen Lookahead)

Syntaktische Prädikate erlauben die Spezifikation eines potentiell unendlichen Lookaheads, der mit einem Backtracking-Algorithmus ausgeführt wird. Ist dieser erfolgreich, wird die entsprechende Alternative ausgewählt. Syntaktische Prädikate verwenden dieselbe Syntax wie eine normale Klammerstruktur, wobei anschließend ein „=>“ genutzt wird. Sie können vor der Angabe von Schnittstellen in einer Regel oder an einer beliebigen Stelle innerhalb eines Regelrumpfs verwendet werden.

Auch in einer nicht-mehrdeutigen Grammatik kann es aufgrund des linearisierten Lookahead-Verfahrens, das Antlr verwendet, zu Mehrdeutigkeiten kommen. Dennoch verwendet Antlr diese Methode zur Auswahl bei Alternativen, da sie auch im schlechtesten Fall ein in Bezug auf den Lookahead lineares Laufzeitverhalten zeigt, wohingegen bei üblichen Recursive-Descent-Parsern ein expontielles Laufzeitverhalten auftreten kann. An Stellen, die von dieser Methode nicht abdeckt werden, können syntaktische Prädikate verwendet werden. Ein Beispiel findet sich in Abbildung 5.20.

MontiCore-Grammatik

```
1 S = ("a" "b" | "b" "a")=> A
2     | B ;
3 A = "a" "b" | "b" "a" ;
4 B = "a" "a" ;
```

Abbildung 5.20.: Syntaktische Prädikate für eindeutige Grammatiken

Semantische Prädikate

Semantische Prädikate sind Java-Ausdrücke, die in geschweiften Klammern und einem abschließenden Fragezeichen ausgedrückt werden. Die Benutzung erlaubt eine gezielte Beeinflussung des Lookaheads oder die Prüfung von Invarianten der Sprache. Dieses Sprachkonstrukt wird nur selten benötigt.

Da der Syntax und die Semantik direkt von Antlr übernommen wurde, verzichten wir hier auf ein Beispiel und verweisen direkt auf die Antlr-Dokumentation (http://www.antlr.org/doc/metalang.html#SemanticPredicates).

Codefragmente

In die MontiCore-Grammatik können an jede Stelle einer Regel Codefragmente in geschweiften Klammern eingefügt werden, die ausgeführt werden, sofern der Parser diese Alternative einer Regel gewählt hat.

Optionsblöcke

Klammerstrukturen können in Antlr mit der Zeichenkette „options { ... }“ beginnen, wobei anstatt der Punkte verschiedene Optionen verwendet werden können. Diese ergeben sich aus der Antlr-Dokumentation (http://www.antlr.org/doc/options.html), wobei meistens „options { greedy=true }“ verwendet wird. Dieses entfernt die Warnungen von Antlr und ändert nichts am Verhalten, weil sich Antlr standardmäßig greedy bei der Auswahl von Alternativen verhält.

Wertübergabe von Variablen

Antlr erlaubt die Übergabe von Variablen an andere Regeln. Dadurch können oftmals syntaktische Prädikate oder eine Umstruktierung der Grammatik verhindert werden. Abbildung 5.21 zeigt die Übergabe von Parametern an andere Regeln.

Innerhalb einer Regel können Variablen an Regeln mittels „->“ übergeben werden. Die Parameterzahl ist dabei nicht beschränkt, wobei die Parameter durch Komma voneinander zu trennen sind.

Die formalen Parameter der Regel werden ebenfalls in eckigen Klammern angegeben. In den Parametern wird dieselbe Notation wie für Nichtterminale verwendet. Dadurch können insbesondere Parameter wiederum als Variablen oder auch direkt als Attribute der AST-Klasse gekennzeichnet werden.

Es ist generell nicht möglich, mehrere Regeln mit demselben Regelnamen und anderen formalen Parametern anzugeben, wie es aus der objektorientierten Programmierung mit Überladung von Methoden möglich ist.

MontiCore-Grammatik

```
1  grammar Variables {
2
3  options {
4          parser lookahead=1
5          lexer lookahead=2
6  }
7
8  Automaton =
9    !"automaton"  Name:IDENT "{"
10     (M=SingleModifier*
11       (States:State->[M] | Transitions:Transition->[M])
12     )* "}";
13
14 State [Modifiers:SingleModifier*] =
15   !"state" Name:IDENT ";" ;
16
17 Transition [Modifiers:SingleModifier*]=
18   From:IDENT "->" To:IDENT ";" ;
19
20 SingleModifier =
21   Public:[!"public"]|Private:[!"private"];
22 }
```

Abbildung 5.21.: Variablenübergabe

5.3.6. Dynamische Einbettung

Werden Grammatiken wie sonst Klassen für die Erstellung von Softwareprojekten genutzt, wollen Entwickler diese schnell wiederverwenden. Diese Wiederverwendung ist schon immer eine Kernidee der Informatik gewesen. Insbesondere kann man eine verbesserte Wiederverwendungsfähigkeit erkennen. Bei Assemblersprachen erfolgte

die Wiederverwendung über Kopieren von Quellcodezeilen, wohingegen bei Hochsprachen die Kapselung von Funktionen bis zur Kapselung von Klassen in objektorientierten Sprachen entwickelt wurde. Die Wiederverwendung erlaubt eine Qualitätssicherung der einzelnen Artefakte und die mehrfache Nutzung in verschiedenen Projekten.

Unter der dynamischen Einbettung einer Grammatik in einen andere versteht man die Verwendung von Nichtterminalen einer Grammatik in einer anderen. Dabei wird in MontiCore nicht direkt angeben, welches andere Nichtterminal und welche andere Grammatik verwendet wird, sondern es wird vielmehr ein Platzhalter definiert. Für diesen Platzhalter wird zur Konfigurationszeit eine Grammatik mit einer Startproduktion eingebunden.

In einer MontiCore-Grammatik wird die dynamische Einbettung durch einen Unterstrich vor einen Nichtterminal gekennzeichnet. Bei der Generierung prüft MontiCore dieses Symbol nicht weiter. Bei der Konfiguration eines eigenen Parsers können nun verschiedene Grammatiken miteinander verbunden werden, indem angegeben wird, welcher Lexer/Parser verwendet werden soll, falls ein solches Symbol geparset werden soll.

MontiCore verfügt über eine Java 5.0 kompatible Grammatik, die z.B. dazu verwendet werden kann, in domänenspezifischen Sprachen Java-Statements einzubetten. Abbildung 5.22 zeigt ein Beispiel für die Einbettung eines JavaBlockStatements, also eines in geschweiften Klammern eingeschlossen Anweisungsblocks. Die Konfiguration des entsprechenden Parsers lässt sich aus Abbildung 5.23 ersehen. Dabei ist zu beachten, dass die Konfiguration im Quellcode erfolgen kann, jedoch meistens nicht muss: Das Konzept „DSLTool“ bietet eine einfache textuelle Syntax an, um die Parserkonfiguration vorzunehmen.

5.4. Konzepte

Eine Grammatik erlaubt die Spezifikation der kontextfreien Syntax einer Sprache und die Generierung eines Parsers mit dazu passenden AST-Klassen. Für die Nutzung einer so entstandenen Sprache ist es oftmals hilfreich, zusätzlich zum Parser eine Infrastruktur zur Verfügung zu haben, um diese schnell in eigene Projekte integrieren zu können. Es ist daher möglich, innerhalb einer Grammtik weitere Informationen anzugeben, die den Generierungsprozeß verändern oder erweitern. Jedes solches Erweiterungsprinzip von Grammatiken wird in MontiCore als Konzept bezeichnet und ähnelt den Pragmas im Compilerbau.

Es existieren in MontiCore bereits einige Konzepte, die bisher aber eher technischer Natur sind. Es ist vielmehr möglich, eigene Konzepte zu entwickeln und auf einfache Weise in MontiCore zu integrieren. Im folgenden werden die existierenden Konzepte und die Entwicklung neuer Konzepte beschrieben.

MontiCore-Grammatik

```
package mc.examples.automaton;

grammar Automaton {

options {        parser lookahead=2 lexer lookahead=3 }

Automaton =
    !"automaton" Name:IDENT "{"
       (States:State | Transitions:Transition)* "}" ;

State =
  !"state" Name:IDENT
  ( "{" (States:State | Transitions:Transition)* "}"
  | ";"
  ) ;

Transition =
   From:IDENT "-" Activate:IDENT ">" To:IDENT
   ("/" BlockStatement:_BlockStatement)? ";" ;
}
```

Abbildung 5.22.: Dynmamische Einbettung von Block-Statements

Java

```
// Create overall parser
overallparser = new MonticoreParser(filename, reader);

// Create Parsers: Automaton
overallparser.addMCConcreteParser(
  new AutomatonAutomatonMCConcreteParser("automaton"));

// Create Parsers: JavaBlockStatement
overallparser.addMCConcreteParser(
  new JavaDSLBlockStatementMCConcreteParser("javablockstm"));

// Add embedding:
// when in Automaton-Parser the rule Block is invoked, the javablockstm-Parser
// (=JavaParser with start rule BlockStatement) is invoked.
overallparser.addExtension("automaton", "javablockstm", "BlockStatement" );


// Set Start parser
overallparser.setStartParser( "automaton" );
```

Abbildung 5.23.: Konfiguration eines MontiCore-Parsers

5.4.1. Globalnaming

Das Konzept „Globalnaming“ erlaubt die Generierung einer flachen Symboltabelle für eine domänenspezifische Sprache. Dabei muss der Entwickler angeben, an welchen Stellen der Grammatik Namen auftreten können. Dabei werden zwei Fälle unterschieden:

Define
: Stellen, an denen ein Name definiert wird und daher nur einmal auftreten darf.

Usage
: Stellen, an denen nur Namen benutzt werden dürfen, die an anderer Stelle mittels `Define` definiert wurden.

Die Stellen der Grammatik werden wie für Konzepte üblich in der Notation «Regelname».«Attributname» definiert.

Ein einfaches Beispiel findet sich in Abb 5.24, in der ein Ausschnitt einer Automatengrammatik dargestellt wird. Weitere Details zur Nutzung und zu den Vorteilen des Konzepts befinden sich im Beispiel „Automaton“ (vgl. Abschnitt 6.1), das mit den MontiCore-Beispielen verfügbar ist.

5.4.2. Classgen

Das Konzept „Classgen“ erlaubt die Definition bestimmter zusätzlicher Attribute und beliebiger Methoden innerhalb der generierten AST-Klassen. Abbildung 5.25 zeigt exemplarisch die Möglichkeiten des Konzepts.

Die Notation des Konzepts ist an das Grammatikformat angelehnt und erzeugt einfache Attribute oder bei angefügtem Stern Listen. Die Konstanten werden auf dieselbe Art wie bei der Grammatik erzeugt, wobei hier zurzeit ein `boolean` oder ein `int` entsteht. Zusätzliche Methoden werden mit dem Schlüsselwort `method` eingeleitet und folgen ansonsten den üblichen Java-Konventionen für eine Methodendefinition. Das Hinzufügen von `toString()`-Methoden erleichtert im Allgemeinen das Debuggen von Algorithmen einer Sprache und wird daher empfohlen.

Sind die Attribute innerhalb einer AST-Klasse bereits vorhanden, hat der im Konzept angegebene Typ Vorrang. Dadurch lässt sich die automatische Typableitung, wie sie in Abschnitt5.3 beschrieben ist, durch einen Nutzer beeinflussen.

Somit lässt sich beispielsweise ein Attribut, das zu einer Liste wird, weil es innerhalb einer Blockstruktur mit kleenschem Stern auftritt, zum einem einfachem Attribut verändern. Der Parser weist dabei das letzte Auftreten, dem Attribut zu. Die Vorgänger werden ignoriert. Spezifiziert man zusätzlich eine maximales Auftreten von eins (`max=1`), erhält der Nutzer bei mehrfachem Auftreten eine entsprechende Fehlermeldung.

MontiCore-Grammatik

```
1  package mc.examples.automaton;
2
3  grammar Automaton {
4
5  concept globalnaming {
6          define State.Name;     // Hier werden Namen zuerst definiert, die
7          usage Transition.From; // hier
8          usage Transition.To;   // und hier benutzt werden können.
9  }
10
11 Automaton =
12     !"automaton" Name:IDENT "{"
13        (States:State | Transitions:Transition)* "}" ;
14
15 State =
16   !"state" Name:IDENT
17   ( "{" (States:State | Transitions:Transition)* "}" | ";" ) ;
18
19 Transition =
20    From:IDENT "-" Activate:IDENT ">" To:IDENT
21    ("/" BlockStatement:_BlockStatement)? ";" ;
22 }
```

Abbildung 5.24.: Einbettung des Globalnaming-Konzeptes

5.4.3. DSLTool

Das Konzept „DSLTool“ erlaubt die Generierung von Hilfsklassen zur Verwendung der Grammatik im DSLTool-Framework. Diese Klassen können stets selbst geschrieben werden, es können aber einfache Standards generiert werden, die üblicherweise ausreichend oder durch Subklassenbildung anpassbar sind.

RootObjekte dienen als Repräsentation einer Eingabedatei innerhalb des Frameworks. Der Oberklasse wird keine Funktionalität hinzugefügt, sondern eine leere Klasse erzeugt, die für das Framework nötig ist. Nach dem Schlüsselwort „root“ folgt ein Name, der einen Klassennamen im selben Paket wie die Grammatik bezeichnet. Danach wird in spitzen Klammern die Grammatikregel angegeben, die als Startregel verwendet werden soll und damit den Typ des ASTs bestimmt. Da die folgenden RootFactories und ParsingWorkflows spezifisch für eine Root-Klasse sind, wird die Notation dort wiederholt.

RootFactories zur Erzeugung von Root-Objekten innerhalb eines DSLTools. Die Root-Objekte werden zusammen mit ihrem Parser erzeugt. Dabei kann spezifiziert werden, welche Parser verwendet werden sollen, und wie diese ineinander eingebettet sind. Eine detailierte Erklärung der Einbettungsmechanismen von MontiCore findet sich in Abschnitt 5.3.6.

MontiCore-Grammatik

```
1 // Erzeugung zusätzlicher Attribute in der Klasse ASTRule
2 concept classgen Rule {
3   // Erzeugt ein Attribut ExtraComponent vom Typ ASTComponent
4   ExtraComponent:Component;
5
6   // Erzeugt ein Attribut ExtraComponents vom Typ ASTComponentList
7   // Beim Parsen wird überprüft, dass mindestens zwei Elemente enthalten sind
8   ExtraComponents:Component* min=2 max=*;
9
10  // Erzeugt ein Attribut ExtraBool vom Typ boolean
11  ExtraBool:[];
12
13  // Erzeugt ein Attribut ExtraInt vom Typ int
14  ExtraInt:[*];
15
16  // Fügt Methode hinzu
17  method public String toString() { return leftSide; } ;
18 }
```

Abbildung 5.25.: Erzeugen von zusätzlichen Attributen und Methoden

Die Parser werden zunächst mit qualizierten Namen bezeichnet, der sich durch einen Punkt voneinander getrennt aus dem qualifizierten Namen der Grammatik und der Startregel zusammensetzt. Danach folgt ein einfacher Bezeichner, der für diesen Parser innerhalb der Factory verwendet wird. Ein weiterer Parser kann in diesen Parser eingebettet werden, indem nach der Definition mittels des Schlüsselworts „in“ ein einfacher Bezeichner und eine Parserregel angeben werden (durch einen Punkt getrennt). Optional wird in Klammern ein Parameter hinzugefügt. Einer der registrierten Parser kann mittels des Stereotypen «start» als Startparser verwendet werden. Diese Form der Einbettung wird im Beispiel zu Abschnitt 6.2 näher erläutert.

ParsingWorkflows dienen zur Ausführung des in einer RootFactory instanzierten Parsers. Der Workflow erlaubt die Generierung des ASTs und bricht bei schwerwiegenden Fehlern die weitere Bearbeitung ab.

Ein Beispiel für eine DSLTool-Konzept befindet sich in Abbildung 5.26. Dabei wird innerhalb der RootFactory ein Parser instanziert, der die Startregel Automaton der Grammatik mc.examples.automaton.Automaton aufruft. Wird beim Parsen die Regel Activate verwendet, wird dafür die eingebettet Java-Grammatik mc.java.JavaDSL verwendet, wobei hier die Regel Expression als Startegel fungiert.

Die vollständige Grammatik zur Spezifikationen eines DSLTools befindet sich in Abbildung 5.27.

MontiCore-Grammatik

```
concept dsltool {

  root AutomatonRoot<Automaton>;

  parsingworkflow AutomatonParsingWorkflow for AutomatonRoot<Automaton> ;

  rootfactory AutomatonRootFactory for AutomatonRoot<Automaton> {
    mc.examples.automaton.Automaton.Automaton aut <<start>> ;
    mc.java.JavaDSL.Expression javaexp in aut.Activate;
  }
}
```

Abbildung 5.26.: Erzeugen von Hilfsklassen für die DSLTool-Infrastruktur

5.4.4. Antlr

Das Konzept Antlr erlaubt die Integration von Java und Antlr-Quellcode in die Grammatik. Diese Fragmente können wahlweise in den Lexer oder den Parser-Code eingefügt werden. Somit können spezielle Konstrukte, die durch das MontiCore-Grammatikformat eventuell nicht unterstützt werden, eingefügt und so eine Modifikation des generierten Quellcodes verhindert werden. Abbildung 5.28 zeigt die Syntax für die Verwendung.

5.4.5. GrammarExclude

Die Verwendung des Konzepts „Antlr“ bedingt eventuell, dass für einige Regeln und Nichtterminale kein Quellcode generiert werden soll. Das Konzept „GrammarExclude“ erlaubt diesen Ausschluss. Die Syntax befindet sich in Abbildung 5.29.

Dabei können nach nach dem Schlüsselwort „rule“ beliebig viele Regeln und nach dem Schlüsselwort „terminal“ beliebig viele Terminaldefinitionen von der Codegenerierung ausgeschlossen werden. Die Terminale werden wie in der Grammatik üblich bezeichnet also z.B. `","` für ein Komma. Terminale, die mit der Definition eines Identifiers übereinstimmen, wie z.B. `!"beispiel"` müssen nicht ausgeschlossen werden.

5.4.6. Entwicklung eigener Konzepte

MontiCore ist auf die Entwicklung eigener Konzepte ausgelegt, um die vorhandene Kernfunktionalität zu erweitern.

MontiCore-Grammatik

```
1  ConceptDsltool =
2    !"concept" "dsltool" "{"
3      ( Roots:Root | ParsingWorkflows:ParsingWorkflow |
4        Rootfactories:RootFactory)* "}" ;
5
6    Root=
7      !"root" RootClass:IDENT "<" Rule:IDENT ">" ";";
8
9    ParsingWorkflow        =
10     !"parsingworkflow" Name:IDENT !"for" RootClass:IDENT "<" Rule:IDENT ">" ";";
11
12   RootFactory        =
13     !"rootfactory" Name:IDENT !"for"  RootClass:IDENT "<" Rule:IDENT ">"
14     "{" (Parsers:Parser)* "}";
15
16   Parser =
17     (Package:IDENT ( "." Package:IDENT)*)? "." DSLName:IDENT "." Rulename:IDENT
18     Name:IDENT (START:["<<start>>"])?
19     (!"in" (Extensions:ParserExtension
20            |ExtensionsWithParameter:ParserExtensionWithParameter)
21            ("," 
22             (Extensions:ParserExtension
23             |ExtensionsWithParameter:ParserExtensionWithParameter))*
24            )? ";" ;
25
26   ParserExtension =
27     Outer:IDENT "." Ext:IDENT;
28
29   ParserExtensionWithParameter =
30     Outer:IDENT "." Ext:IDENT "(" Parameter:IDENT ")";
```

Abbildung 5.27.: Grammatik des DSLTool-Konzepts

Die Konzepte können frei entworfen werden und unterliegen keinen Beschränkungen. Damit sie jedoch stilistisch zu bereits vorhandenen Konzepten und dem MontiCore-Grammatikformat passen und somit ein Nutzer sie leicht verwenden kann, sollten folgende Konventionen eingehalten werden:

- Die Startregel des Konzeptes X heißt ConceptX und beginnt mit dem Schlüsselwort x (klein!).
- Danach folgt der Körper des Konzepts in geschweiften Klammern.
- Beziehen sich Elemente auf Komponenten in Regeln, dann werden diese wie folgt geschrieben: „Regelname.Komponentenname“.

EBNF

```
1 AntlrConcept ::=
2   '{'
3       ( AntlrParserCode | AntlrParserAction
4       | AntlrLexerCode  | AntlrLexerAction  )*
5   '}' ;
6
7 AntlrParserCode ::=
8   'parser' 'antlr' '{' <<Antlr-Parser-Regeln>> '}' ;
9
10 AntlrParserAction ::=
11   'parser' 'java' '{' <<Java-Code für den Parser>> '}' ;
12
13 AntlrLexerCode ::=
14   'lexer' 'antlr' '{'<<Antlr-Lexer-Regeln>> '}' ;
15
16 AntlrLexerAction ::=
17   'lexer' 'java' '{' <<Java-Code für den Lexer>> '}' ;
```

Abbildung 5.28.: Verwendung von Antlr- und JavaCode in einer MontiCore-Grammatik

EBNF

```
1 GrammarExclusion ::=
2   'concept' 'grammarexclude' '{' (Rule | Terminal )* '}' ;
3
4 Terminal ::=
5   'terminal' STRING (',' STRING )* ';' ;
6
7 Rule ::=
8   'rule' IDENT (',' IDENT )* ';' ;
```

Abbildung 5.29.: Verwendung des Grammarexclude-Konzepts

6. Weitere Beispiele

MontiCore ist an einer Reihe weiterer Beispiele getestet worden. In diesem Kapitel finden sich kurze Beschreibungen solcher Beispiele, die das Tutorial aus Kapitel 2 um weitere Aspekte ergänzen. Die Beispiele sind über denselben Mechanismus im MontiCore-Eclipse-Plugin[1] verfügbar. Dazu muss unter `File>New ...` ein neues MontiCore-Projekt angelegt und im Wizard das jeweilige Beispiel ausgewählt werden. Im Anhang A befinden sich Screenshots der wichtigsten Schritte.

Die Beispiele sind nicht sofort kompilierbar, da der MontiCore-Generator zunächst die entsprechenden Klassen generieren muss. Dafür gibt es die folgenden drei Methoden:

Generate.java
: Die Klasse Generate lädt bei Ausführung der main-Methode eine MontiCore-Instanz und führt den Generierungsprozeß aus. Der Eclipse-Java-Compiler kompiliert anschließend automatisch die Quelldateien.

build.xml
: Die Ant-Builddatei ruft MontiCore auf und generiert so die entsprechenden fehlenden Klassen. Durch diese Methode wird auch gleich der komplette Quellcode kompiliert (Aufruf: `ant compile`).

Nature
: Durch die Aktivierung der MontiCore-Nature werden die MontiCore-Eingabedateien automatisch zur Generierung verwendet. Der Eclipse-Java-Compiler kompiliert anschließend automatisch die Quelldateien.

Die Beispiele sind ebenfalls ohne die Verwendung von Eclipse verfügbar, da aus der `examples.jar` der entsprechende Beispielquellcode extrahiert werden kann:
`java -jar examples.jar «DSL» «outputdir»`.
Dabei entspricht `«DSL»` dem DSL-Namen des Beispiels und `«outputdir»` dem gewünschten Ausgabeverzeichnis. Ein Aufruf ohne Parameter listet die verfügbaren Beispiele auf.

[1] Die Beispiele sind unter www.monticore.de verfügbar. Im folgenden wird das Archiv unabhängig von dessen Version mit `examples.jar` bezeichnet

In den folgenden Abschnitten werden jeweils kurz die Aspekte des MontiCore-Frameworks erwähnt, die mittels des jeweiligen Beispiels näher erklärt werden sollen. Eine detaillierte Beschreibung ergibt sich aus der Dokumentation des Quellcodes und der beigefügten Readme-Dateien.

6.1. AutomatonDSL

Die AutomatonDSL beschreibt eine einfache DSL, die endliche erkennende Automaten um Hierarchie erweitert. Die Automaten können einen Eingabestrom von Zeichen verarbeiten, wobei eine Transition immer genau ein Zeichen verarbeitet und es somit keine Epsilon-Transitionen gibt. Diese Form von Automat muss eindeutig sein, d.h. von einem Zustand ausgehend darf es nur eine Transition mit derselben Aktivierung geben. Diese Form der Automaten befindet sich in nur einem einzigen Zustand, verhält sich also anders als Statecharts, bei denen sofort initiale Unterzustände ebenfalls betreten werden. Daraus ergibt sich, dass diese Form der Automaten nur einen globalen initialen Zustand haben kann. Bei der Ausführung werden jedoch auch die Oberzustände eines Zustand beachtet: Ausgehende Transitionen von einem Oberzustand werden vorrangig vor denen im aktuellen Zustand geschaltet.

Im diesem Beispiel für eine DSL wurde ein GlobalNaming (vgl. Abschnitt 5.4) verwendet, um die Namen der Zustände mit den Bezeichnungen der Transitionen logisch zu verbinden. Dadurch kann automatisch geprüft werden, ob in den Transitionen nur Bezeichner verwendet werden, die als Zustand definiert sind. Auf dieser Infrastruktur aufbauend können die Zustände automatisch umbenannt werden, ohne dieses Refactoring spezifisch für diese DSL zu programmieren (vgl. `mc.examples.automaton.renaming.RenameAndPrintWorkflow`). Zusätzlich zur wird mit Hilfe der MontiCore-Transform-Klassen gezeigt, wie sich die Ausführung einer solchen DSL einfach entwickeln lässt (vgl. `mc.examples.automaton.execute.ExecuteWorkflow`).

6.2. EmbeddingDSL

Mit Hilfe der EmbeddingDSL wird gezeigt, wie sich verschiedene Sprachen ineinander einbetten lassen. Die „äußere“ Sprache ist eine einfache Variante von Statemachines. Darin eingebettet sind „Assignments“ einer Sprache „Action“, die sich zum einen als Aktionen an Transitionen und zum anderen als „entry“ Aktionen von Zuständen notieren lassen. Als alternative Sprache zur Einbettung von Aktionen in Zustände stehen Java Block Statements zur Verfügung. Dabei ist es möglich, beide Sprachen innerhalb einer Instanz einzubetten (siehe `input\embedding\A.sm`). Sowohl die erste Variante der Einbettung genau einer Sprache an eine Stelle in der Grammatik, als auch die zweite Variante, der parametrischen Einbettung mehrerer Sprachen an

derselben Stelle in einer Grammatik, lassen sich in anhand der Grammatikdefinition `mc.examples.embedding.outer.StateMachine` nachvollziehen.

Das Beispiel zeigt weiterhin die Implementierung eines PrettyPrinters, also eine wohlformatierte Ausgabe der Eingabedatei, für eingebettete Sprachen (vgl. `mc.examples.embedding.prettyprint`).

6.3. PetrinetDSL

Dieses Beispiel demonstriert die Entwicklung einer Sprache zur Beschreibung von Petrinetzen und die Unterstützung durch das MontiCore-Framework. Insbesondere soll hier demonstriert werden, wie syntaktische Analysen[2] auf einer Eingabe vorgenommen werden können.

Petrinetze können als gerichtete Graphen aufgefasst werden, für die zwei Knotentypen zugelassen sind: Transitionen und Stellen. Dabei dürfen Stellen nur mit Transitionen verbunden werden. Stellen dürfen in einem boolschen Perinetz einen, in einem numerischen Petrinetz mehrere Marker bzw. Token enthalten. Optional können bei der letzteren Variante sowohl die maximale Anzahl von Tokens pro Stelle als auch eine Kardinalität pro Kante (maximal konsumierte Tokens pro Schritt) festgelegt werden.

Nach dem Einlesen einer textuellen Repräsentation durch den generierten Parser lassen sich auf dem erzeugten AST weitere Operationen durchführen. Als Beispielanwendung werden in diesem Abschnitt exemplarisch folgende syntaktische Überprüfungen durchgeführt, die kontrollieren, ob ein valides Petrinetz vorliegt (vgl. `mc.examples.petrinet.analysis.SyntacticAnalysis`):

- Bei boolschen Petrinetzen wurden für alle Stellenbelegungen entweder `true` oder `false` verwendet.
- Bei numerischen Petrinetzen gibt es für alle Stellenbelegungen nur nummerische Angaben.
- Bei boolschen Petrinetzen besitzen keine Kante eine Kardinalität.
- Die initiale Belegung einer Stelle überschreitet nicht ihr Maximum.

Des Weiteren verdeutlicht dieses Beispiel die Integration eines solchen Workflow zur syntaktischen Analyse in einen möglichen Gesamtablauf. Ein PrettyPrinter für ein

[2] Im Compilerbau ist der Begriff syntaktische Analysen auf das Parsen der Eingabe festgelegt. Darüber hinausgehende Analysen werden als semantische Analysen bezeichnet. In unserer Terminologie ist Semantik für viel weitreichendere Konzepte vorbehalten, so dass wir an dieser Stelle von syntaktischen Analysen sprechen wollen.

Petrinetz wird nur aufgerufen, falls die Überprüfung ohne Fehler beendet wird. Liefert die syntaktische Überprüfung einen Fehler, wird der Gesamtprozess mit einem fatalen Fehler abgebrochen
(vgl. `mc.examples.petrinet.prettyprint.PetrinetCheckAndPrintWorkflow`).

7. Verwandte Arbeiten

7.1. EMF

Das Eclipse Modeling Framework (EMF) [BSM+04, WWWe] bezeichnet ein quelloffenes Modellierungsframework, das die Basis für Applikationen bildet. Insbesondere verwendeten einige Werkzeuge zur Erstellung von domänenspezifsichen Sprachen EMF und das im Folgenden erklärte GMF zur Erstellung von Metamodellen und grafischer Instanzeditoren.

Ein EMF-Metamodell kann direkt in der Entwicklungsumgebung Eclipse erstellt werden, wobei dazu ein einfacher hierachischer Editor verwendet wird. Alternativ kann ein Metamodell auch aus einem UML-Klassendiagramm, annotierten Java-Schnittstellen oder einem XML-Schema abgeleitet werden. Mit Emfatic [WWWf] kann das Metamodell zuzätzlich auch textbasiert spezifiziert werden. Dieses Metamodell wird als Kernmodell bezeichnet, aus dem automatisch ein Generierungsmodell abgeleitet werden kann. Dieses Modell kann der Nutzer dann gezielt um spezifische Implementierungsinformationen angereichern. Aus diesem Modell lassen sich Java-Klassen generieren, die spezielle Tags kennzeichente Lücken enthalten, damit der Nutzer einerseits die Klassen um Verhalten ergänzen kann und andererseits dieser Quellcode auch bei einer erneuten Generierung erhalten bleibt. Zusätzlich zur Klassengenerierung wird eine Objektserialisierung in ein XML-Format, Fabrikklassen und ein einfacher Instanz-Editor für Eclipse generiert.

Das verwendete Metamodell von EMF wird mit Ecore (vgl. Abbildung 7.1) bezeichnet und orientiert sich am UML-Metamodell, beschränkt sich jedoch auf die Modellierung der statischen Aspekten von Modellen. Dabei wird nur ein Teil der UML-Möglichkeiten genutzt, deren Implementierung in Java einfach möglich ist. Ergänzend zum UML werden implementierungsnahe Details wie Fabriken zur Objekterzeugung und URIs zur eindeutigen Bezeichnung von Paketen in das Metamodell aufgenommen.

Eine eigene Constraint-Sprache um weitergehende Einschränkungen für die Modelle auszudrücken, wie die OCL für UML/MOF-Modelle, ist im EMF-Kern nicht vorgesehen. Es gibt jedoch Ergänzungen wie das EMF OCL Plugin[WWWg], die ein solches Vorgehen erlauben. Innerhalb des Eclipse Modeling Project [WWWh] ist ebenfalls eine Integration der OCL für EML-Modelle geplant.

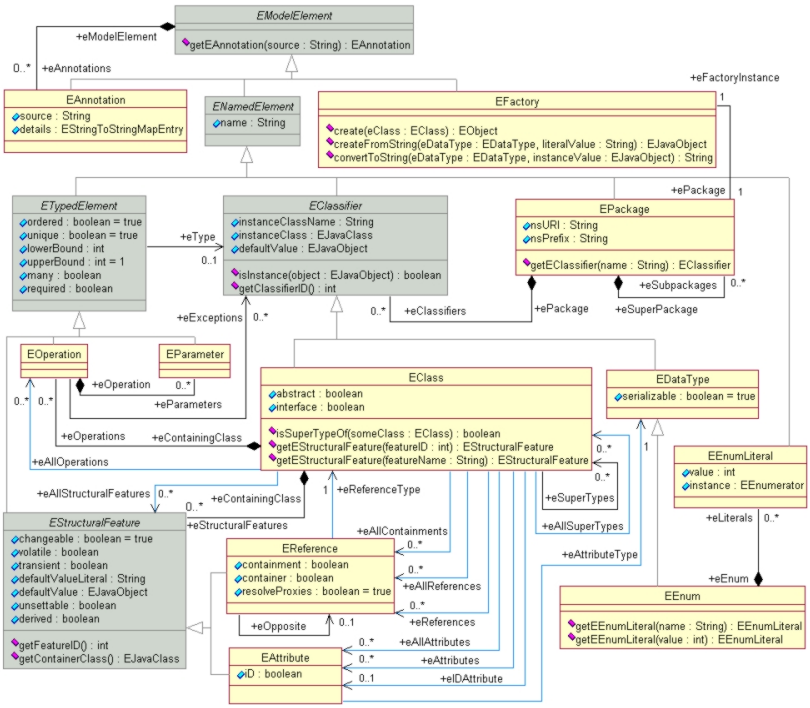


Abbildung 7.1.: Das Metamodell von EMF als UML-Klassendiagramm: Ecore [WWWp]

7.2. GMF

Das Graphical Modeling Framework (GMF) [WWWk] baut auf EMF auf und erlaubt die Generierung grafischer Instanzeditoren für EMF-Modelle mittels des Grahical Editing Frameworks (GEF) [WWWi]. Diese ersetzen die einfachen Editoren, die EMF standardmäßig erzeugt.

Dazu müssen zum Metamodell, das die Domäne beschreibt und daher im Zusammenhang von GMF als Domain Model Definition (DMD) bezeichnet wird, drei zusätzliche Modelle erzeugt werden: Die Graphical Definition (GD), die Tooling Definition (TD) und die Mapping Definition (MD). Aus diesen drei Beschreibungen ergibt sich ein EMF-Generator-Modell, das sich wie bei EMF üblich nochmals anpassen lässt, und schließlich die Grundlage für das Editor-Plugin bildet. Eine Übersicht ist in Abbildung 7.2 dargestellt.

Die GD beschreibt die grafische Darstellung einzelner Elemente der Modelle. Dabei können vordefinierte Elemente von GMF wiederverwendet werden oder eigene Grafiken eingebunden werden. Die Elemente der Modelle werden dabei auf die drei Grundtypen Klassen, Attributen dieser Klassen und Verbindungen zurückgeführt und entsprechend dargestellt. Die TD beschreibt die Konfiguration der Benutze-

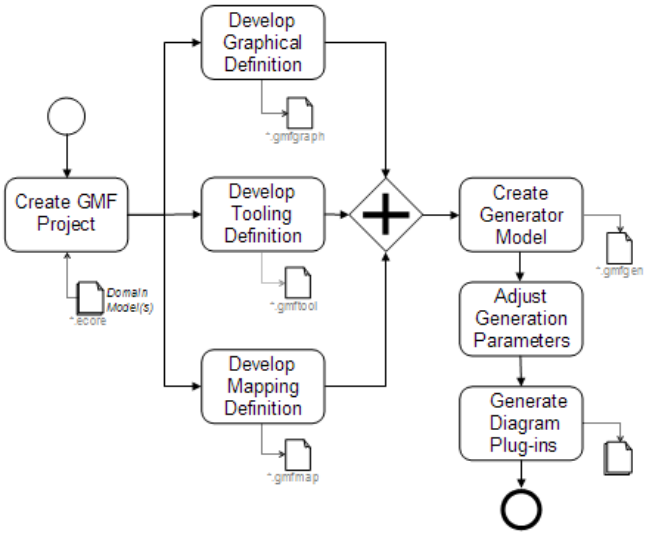


Abbildung 7.2.: Übersicht über den Generierungsprozess von GMF [WWWl]

roberfläche des Instanzeditors. Dabei kann u.a. die Liste der durch einen Nutzer erzeugbaren Objekte festgelegt und die Struktur der Menüleisten bestimmt werden. Die MD ist ein sehr einfaches Modell, das im Wesentlichen die drei Modelle DMD, GD und TD referenziert und für den folgenden Generierungsprozeß zusammenfasst.

7.3. Kermeta

Kermeta[WWWr] ist quelloffenes Projekt der Triskell-Arbeitsgruppe unter der Leitung von Prof. Jean-Marc Jézéquel. Das Triskell Projekt setzt sich aus Wissenschaftlern der Einrichtung IRISA (Forschungseinheit von CNRS, Université Rennes, INRIA und INSA) zusammen.

Kermeta ist eine Metamodell-Sprache, die Verhalten und Struktur von Modellen beschreiben kann. Sie ist dabei kompatibel für EMOF (Essential MOF) und Ecore. Sie soll die Basis für Aktions-, Transformations-, Constraint- und Metadata-Sprachen bilden.

Kermeta wird von den Entwicklern als modellorientoiert beschrieben, da es einem Nutzer leicht ermöglicht, durch Modelle zu navigieren. Die grundlegende Struktur der Sprache ist impertativ und objektorieniert. Kermeta verwendet dabei eine generisches Typsystem, erlaubt Mehrfachvererbung, jedoch keine Operationsüberladung. Es können Funktionspointer (lambda Ausdrücke) genutzt werden. Die derzeitige Implementierung verwendet einen Java-basierten Interpreter und unterstützt den Benutzer in der Eclipse IDE durch einen Editor und Debugger. Kermeta verfügt über

eine einfache Klassenbibliothek, die Standardausgabenermöglicht und Persistenz von Modellen (Laden und Speichern) ermöglicht.

Kermeta kann Ecore-Metamodelle/Modelle verarbeiten. Die eigenen Kermeta-Modelle stellen ein Erweiterung von Ecore-Modellen dar, weil sie zusätzlich Verhalten spezifizieren. Daher kann mit den üblichen Werkzeugen (EMF umd GMF) ein Modell und ein Instanzeditor erstellt werden. Kermeta bietet hierfür keine weitere Unterstützung an.

Kermeta erlaubt die Codegenerierung aus den Modellen mit der Programmiersprache selbst, stellt aber keine über eine reine Ausgabe hinausgehende Funktionalität zur Verfügung. Aus Kermeta heraus ist es jedoch möglich, statische Java-Methoden aufzurufen, so dass eine Einbindung einer Template-Engine wie Velocity möglich ist. Auf dieselbe Art und Weise können ebenfalls andere Werkzeuge angebunden werden.

Kermeta ist textuell und Datei-orientiert und kann damit übliche Versionverwaltungen für Programmiersprachen nutzen.

7.4. MetaEdit+

MetaEdit+ ist ein kommerzielles Produkt von MetaCase[WWWv] zur Definition von graphischen Modellierungssprachen. Es gliedert sich in zwei verschiedene Komponenten: der Method Workbench zur Definition des Metamodells und MetaEdit+ als Tool zur Anwendung der Modellierungssprache.

Das Metametamodell von MetaEdit+ besteht im Wesentlichen aus 6 Grundkonzepten:

1. Properties zur Definition von Attributen und deren Typ (z.B. Attribut „name“ als String).
2. Objects zur Definition von Objekten, die über Properties verfügen können (z.B. „Klasse“ mit dem Attribut „name“).
3. Relations zur Definition von Beziehungen zwischen Objekten (z.B. Vererbung in Klassendiagrammen).
4. Roles zur Definition der Rollen von Objekten in einer Beziehung (z.B. Super- und Subklasse bei einer Vererbung).
5. Ports zur Definition von Schnittstellen der Objekte zur Anbindung bestimmter Beziehungen (bekannt aus Kompositionsstrukturdiagrammen).
6. Graphs zur Definition des Aufbaus eines gültigen Modelles, insbesondere werden hier Constraints für Beziehungen (erlaubte Partner, Kardinalität etc.) festgelegt.

Die Erstellung der Konzepte und Constraints erfolgt menübasiert, für jedes Konzept existiert ein eigener Dialog zur Definition. Für Objects, Relationships und Roles können Graphiken festgelegt werden, die im später resultierenden Werkzeug zur Modellierung Anwendung finden. In diesem befinden sich in der Menüleiste entsprechende Knöpfe zur Erstellung der Komponenten, die dann in einer Editorfläche gezeichnet werden, benötigte Properties werden durch Dialoge abgefragt und anschließend entsprechend gesetzt.

Die Navigation durch ein konkretes Modell und mögliche Codegenerierung erfolgt durch eine eigene Report Definition Language (RDL). Diese beinhaltet unter anderem Kontroll- und Iterationsanweisungen wie `foreach`, `do`, `if`, sowie Funktionen zum Zugriff auf Dateien. Die Möglichkeit zur Definition eigener Reports unterstützt die Codegenerierung durch Iteration über das zugrundeliegende Modell. Die Modellierung und der Zugriff auf das Modell kann jedoch nur innerhalb vom MetaEdit+ erfolgen, eine Interoperabilität mit anderen Werkzeugen und die Möglichkeit zur Erstellung eines Standalone-Werkzeuges ist nicht gegeben. Die Verwaltung des Metamodells und des Modells wird vollständig intern, d.h. nicht auf Dateiebene geregelt, eine Nutzung einer Versionsverwaltung wie CVS ist damit nicht möglich. Auch MetaEdit+ bietet selbst keinerlei Unterstützung der Teamarbeit.

7.5. GME

Das Generic Modeling Environment (GME, [LMB+01]) wurde an der Vanderbilt University am „Institute for Software Integrated Systems“ entwickelt und steht zurzeit in der Version 6.5.8 unter der Apache Software Lizenz frei zur Verfügung [WWWj]. Das Werkzeug erlaubt die Spezifikation von graphischen DSLs mit Hilfe von MetaGME, einer auf UML Klassendiagrammen basierenden und ebenfalls graphischen Metamodellierungssprache. Die Metaklassen wurden um verschiedene Stereotypen erweitert, um Elemente der DSL und deren Beziehungen beschreiben zu können (etwa `«Model»` für die Definition eines Sprachelements oder `«Connection»` zur Spezifikation einer möglichen Assoziationsbeziehung zwischen Elementen). Über ein Eigenschaftenfenster können die Metamodell-Elemente konfiguriert werden, um so die Darstellung der DSL und andere Eigenschaften festzulegen. Vier Karteireiter bieten darüber hinaus unterschiedliche Sichten auf das Modell, wobei die Konsistenz zwischen diesen Sichten vom Werkzeug sichergestellt wird:

- ClassDiagramm: Das Metamodell der DSL als Klassendiagramm. Zwischen den Metaklassen sind Kompositions- und Vererbungsbeziehungen ohne Mehrfachvererbung möglich.
- Visualization: In diese Ansicht kann bestimmt werden, welche Metaklassen sichtbare Elemente der DSL spezifizieren.

- Constraints: Für Kontextbedingungen stehen spezielle Symbole zur Verfügung, die über Assoziationen den Metaklassen zugeordnet werden können. Die Kontextbedingungen selbst werden textuell mit Hilfe der Object Constraint Language (OCL) im Eigenschaftenfenster des Kontextsymbols definiert.
- Attributes: Spezifikation von Attributen der DSL-Elemente. Es stehen `Boolean`, `Enumeration` und `Field` zur Verfügung.

Nach der Interpretation des Metamodells wird die spezifizierte DSL automatisch in GME als Sprache integriert und kann fortan bei der Erstellung eines neuen Projektes ausgewählt werden. In GME sind damit Erstellung und Nutzung einer DSL in einem Werkzeug zusammengefasst. Um die Wiederverwendbarkeit zu erhöhen, bietet GME darüber hinaus die Kombination von verschiedenen Modellierungssprachen und Modellbibliotheken an, wobei die Modelle im XML-Format gespeichert werden.

Die Codegenerierung zu einer in GME erstellten DSL kann in jeder Sprache geschrieben werden, die die COM-Schnittstelle von Microsoft unterstützt. Beispiele hierfür sind C++, Visual Basic, C# und Python. Für die Erstellung einer entsprechenden Projektvorlage für den Zugriff auf die Objekte der DSL steht im Unterordner `SDK/` des GME-Programmverzeichnis das Werkzeug *CreateNewComponent.exe* zur Verfügung. Der Zugriff auf die DSL-Elemente ist dabei über eine vom Typ des jeweils zugehörigen Metamodellelements abhängigen festen Schnittstelle gelöst. Die implementierte Codegenerierung kann schließlich in GME aus den auf der DSL basierenden Projekten heraus gestarted werden.

7.6. DSL-Tools

Die Domain-Specific Language Tools (DSL-Tools) ist ein von Microsoft in der Entwicklung befindlicher Ansatz für die Erstellung von grafischen domänenspezifischen Sprachen. Sie sind ein Teil des Visual Studio SDKs, dass nach einer Registrierung für das Visual Studio Industry Partner Programm (VSIP) frei unter [WWWc] zur Verfügung steht. Für die Nutzung wird außerdem Visual Studio 2005 Professional vorrausgesetzt.

Nach der Installation des SDKs kann unter Visual Studio der Domain Specific Language Designer genutzt werden, um ein neues Projekt für die Definition des Metamodells zu erstellen. Dabei stehen mit Aktivitäts-, Klassen- und Use-Case-Diagrammen, sowie einer sogenannten „Minimal Language“ vier Metamodelle als Vorlagen zur Verfügung, die als Ausgangspunkt für das eigene Metamodell dienen können. Das Metamodell selbst wird über eine eigene, an Klassendiagramme angelehnte Modellierungssprache definiert, die folgende Elemente enthält:

- Class: Klasse zur Definition eines Sprachelements

- Value Property: Eigenschaftswerte (Attribute) von Sprachelementen, die Klassen und Beziehungen zwischen Klassen zugeordnet werden können.

Darüber hinaus gibt es drei Arten von Beziehungen zwischen Klassen:

- Embedding: Kompositionsbeziehung
- Reference: Assoziationsbeziehung
- Inheritance: Vererbungsbeziehung, wobei Mehrfachvererbung nicht zugelassen ist

Jedes Element enthält einen Konfigurationsbereich, in dem elementspezifische Eigenschaften wie Kardinalitäten oder Sichtbarkeiten festgelegt werden können.

Für die Entwicklung einer domänenspezifischen Sprache werden im wesentlichen drei Dateien benötigt. In *DomainModel.dsldm* wird das Metamodell grafisch per Drag and Drop aus dem Domain Model Designer heraus spezifiziert. Das spätere Aussehen der so definierten Elemente der DSL ist in *Designer.dsldd* in einer XML-Syntax enhalten, wobei Ressourcen wie Bilder, Nachrichten oder Bezeichnungen der DSL-Elemente in *Designer.Resource.resx* aufgeführt sein müssen. Konstraints auf dem Metamodell können in einer zusätzlichen Datei in C# spezifiziert werden. Ein Refactoring zwischen diesen Dateien ist nicht vorhanden, so dass der Anwender selbst für die Sicherung der Konsistenz sorgen muss. Die Überprüfung der DSL geschieht über eine eigene Test-Instanz von Visual Studio, die derekt aus dem Metamodell heraus erzeugt werden kann.

Für die Codegenerierung steht ein Template Transformation Toolkit zur Verfügung, das die Erstellung von Templates in C# und Visual Basic mit iterativen Zugriff auf die aus den DSL-Elementen erzeugten .NET Framework Klassen erlaubt. Obwohl in Visual Studio integriert, wird bei der Template-Erstellung keinerlei Unterstützung wie Syntaxhighlighting oder kontextsensitive Hilfe geboten.

Um die erstellte DSL in anderen Projekten zu nutzen, kann eine Installationsdatei erzeugt werden, die die Integration in Visual Studio erlaubt.

7.7. MPS

Das Meta Programming System (MPS) ist eine Initiative von JetBrains [WWWw] zur Entwicklung von Modellierungssprachen und zugehörigen Editoren. JetBrains hat einen großen Bekanntheitsgrad durch ihre Erfahrungen im Gebiet der Entwicklung von IDEs. Ihr IntelliJIDEA ist eine Entwicklungsumgebung, welche vor allem Java- und Webprogrammierung (u.a.HTML, CSS, JavaScript, JSP) unterstützt.

Voraussetzung für MPS ist eine laufende Version von IntelliJIDEA. Einerseits wird hier entstandener Code dargestellt, andererseits wird sie für Java-spezifische Aufgaben benutzt, wie etwa das Compilieren von Code oder das Laden von Bibliotheken.

Die Eingabe im Meta Programming System wird nicht wie gewohnt, durch reinen Text oder Graphik realisiert, sondern durch das Ausfüllen von Zellen und Eingabeboxen. Sowohl das Metamodell, als auch die Definition des zugehörigen Editors und Modellinstanzen werden durch dieses Konzept erstellt.

Das Metametamodell bietet im Wesentlichen 3 Kernkonzepte:

1. Properties zur Beschreibung von Attributen und deren Typ.
2. Concepts sind vergleichbar mit Klassen, verfügen also über Properties und Links zu anderen Concepts.
3. Links stellen beziehungen zwischen Concepts dar. Es existieren die Subtypen Aggregation oder Referenz, wobei letztere mit Assoziationen vergleichbar ist. Einem Concept können Links zugeordnet werden, wobei die Möglichkeit zur Definition der Kardinalitäten 0 oder 1, 1, 0 bis n und 1 bis n besteht.

Für jedes Concept muss eine Eingabemaske zur Instanzerstellung definiert werden, hier wird angegeben, welche Properties der Instanz wie definiert werden können. Es stehen verschiedene Möglichkeiten wie Eingabefelder oder Drop-Down-menüs zur Verfügung. Nach Definition des Metamodells und der Eingabemasken kann das Projekt compiliert und neue Instanzen erstellt werden.

Zur Codegenerierung bietet MPS die Möglichkeit zur Definition von Generatoren. Diese können über verschiedene Mappings verfügen, welche die zu transformierende Conceptinstanzen auswählen. Dazu generiert MPS in IntelliJIDEA leere Methoden (sog. Queries), die alle Instanzen eines Concepts im aktuellen Modell übergeben bekommen und die zu Transformierenden zurückgeben. Der Inhalt die Methoden, d.h. die Selektion der zu transformierenden Conceptinstanzen, erfolgt durch die Programmierung durch den Benutzer. Die zurückgelieferten Instanzen können anschließend an selbst definierte Templates übergeben werden. Templates bezeichnen hierbei meist Javaklassen, deren Code parametrisiert ist, um an entsprechenden Stellen mit den Werten von Properties des Concepts besetzt zu werden.

Durch die eigenwillige Eingabe und Definition sowohl von Metamodell, als auch der Modellinstanzen bietet das MPS keeine Interoperabilität mit anderen Werkzeugen. Eine Unterstützung von Versionskontrollen und Teamarbeit existiert nicht.

7.8. LISA

Lisa [WWWt, MLAZ02] ist eine interaktive Umgebung zur Entwicklung von (domänenspezifischen) Sprachen. Der Ansatz basiert auf „attribute grammars“, das heißt

auf mit Attributen angereicherten Grammatikdefinitionen, die eine sofortige Interpretation bzw. Evaluation (Auswertung) einer Instanz erlauben.

Informationen, die im Parsebaum für eine Evaluation vorhanden sein müssen, werden in Attributen gespeichert, wobei zwischen „synthetisierten“ und „geerbten“ Attributen unterschieden wird. Werte von synthetisierten Attributen ergeben sich aus Auswertungsregeln (und auch aus geerbten Attributen) und können somit Informationen durch den Parsebaum nach oben reichen. Geerbte Attribute werden vom Vaterknoten geerbt und lassen Information den Parsebaum hinunter wandern.

Eine neue Sprache wird durch Angabe einer attributierten Grammatik, das heißt einer kontext-freien Grammatik zusammen mit Scanner-Regeln, einer Menge von Attributen und „semantischen Regeln“ zur Auswertung der Attribute, definiert. Bei der Entwicklung einer Sprache kann ein „dependency graph“ zur Veranschaulichung des Attributflusses generiert werden. Ebenfalls steht eine graphische BNF-Ansicht zur Verfügung.

Zusätzlich zur herkömmlichen Definition von attributierten Grammatiken bietet Lisa die Möglichkeit, Templates und Mehrfachvererbung von Grammatiken einzusetzen [MZLA99]. Templates für typische Attributzuweisungsfolgen sollen die Lesbarkeit und Wartbarkeit einer Grammatik erhöhen. Die Mehrfachvererbung verbessert die Erweiterbarkeit und Modularität.

Aus der Grammatikdefinition werden Scanner, Parser und „Evaluatoren“ in Java generiert. Dabei ist die Auswahl verschiedener Parsertechniken und Evaluatorstrategien möglich. Für die Berechnung von Attributen steht Java innerhalb der Grammatik zur Verfügung. Bei der Zuweisung von Werten an Attribute werden Java-Ausdrücke verwendet. Benutzerdefinierte Methoden (beispielsweise um eine Hashtabelle zu verwalten) werden direkt als `method`-Block in die Grammatik integriert.

Als Instanzeditor dient ein integrierter Texteditor, bei dem durch Auswahl eines passenden Scanners, ein Syntaxhighlighting für eine Sprache definiert werden kann. Außerdem kann der Syntaxbaum und eine zellenbasierte „structure view“ für eine konkrete Instanz angezeigt werden. Der Syntaxbaum kann beim Parsen einer Instanz zudem animiert werden.

Eine Instanz kann entweder vollständig automatisch oder Schritt-für-Schritt evaluiert werden, wobei es möglich ist, über einen „evaluator tree“ die aktuellen Attributwerte anzuzeigen.

Parser und Evaluatoren werden in Java generiert und sind auch stand-alone verwendbar. In [MLAZ98] ist gezeigt wie, mit Hilfe von Lisa, Code für eine Sprache generiert werden kann.

Interoperabilität mit anderen Werkzeugen kann durch den Import der generierten Java-Quellen erreicht werden. Die Entwicklung einer Sprache findet aber vollständig in der Lisa-eigenen Umgebung statt. Eine Eclipse-Integration für eine Erweiterung

zur Generierung von Debuggern und weitern „Animatoren“ befindet sich in der Entwicklung [HPM+04].

Die Sprachdefinition als attributierte Grammatik und die Definition von Sprachinstanzen erfolgt textuell. Eine gemeinsame Entwicklung über ein Versionsverwaltungssystem ist aufgrund der textuellen Darstellung daher möglich.

8. Ausblick

Dieser technische Bericht hat ein Toolframework für die agile Entwicklung von domänenpezifischen Sprachen mit dem MontiCore-Framework beschrieben. Dabei wird mittels einer erweiterten Grammatik die konkrete Syntax einer Modellierungssprache beschrieben. Aus dieser Beschreibung können mittels des MontiCore-Generators Komponenten erzeugt werden, die es erlauben, spezifische Werkzeuge für eine Anwendungsdomäne zu entwickeln.

Zusätzlich zu dieser generellen Vorgehensweise wurden die relevanten Eingabesprachen für den Generierunsgprozess erklärt und anhand von Beispielen die Nutzung verdeutlicht. Dabei wurde ein einfaches Beispiel ausführlich erklärt, um Entwicklern einen einfachen Einstieg zu ermöglichen. Erweiterte Möglichkeiten des Frameworks wurden durch Beispiele verdeutlicht, die mit MontiCore ausgeliefert werden.

Aus den Erfahrungen mit dem MontiCore-Framework ergeben sich zahlreiche Erweiterungsmöglichkeiten, die wir in zukünftige Versionen integrieren wollen. Die wesentlichen Erweiterungen werden im Folgenden dargestellt. Ergänzend zum DSLTool-Framework, das zur Erstellung von Codegeneratoren verwendet werden kann, sollen in Zukunft mehrere erweiterte Frameworks entwickelt werden, die alle gemeinsam dieselben Komponenten nutzen können, jedoch verschiedene Arbeitsabläufe unterstützen. Dabei soll zunächst ein AnalyseTool-Framework implementiert werden, das zur Analyse von Softwareentwicklungsprojekten geeignet ist. Die vorgefertigten Analysen können dann leicht durch den Nutzer ergänzt werden, um spezifische Aspekte eines Softwareprojekts analysieren zu können. Zur Realisierung eines solchen Frameworks sind insbesondere automatisch generierte Symboltabellen notwendig.

Die Entwicklung spezifischer Codegeneratoren wird durch Transformationstechniken deutlich vereinfacht. Dabei wird transparenter, wie einzelne Komponenten miteinander interagieren und die Korrektheit der Ergebnisse lässt sich leichter erfassen. Daher wollen wir in Zukunft prüfen, wie sich vorhandene Tranformationstechniken am besten in die DSLTool-Infrastruktur integrieren lassen. Dabei ist auch die Entwicklung einer eigenen Transformationsprache mit MontiCore denkbar.

Eine spezielle Ausprägung der UML [Rum04b, Rum04a] soll die Grundlage für einen Codegenerator bilden. Dabei beschreiben die UML-Modelle Eigenschaften einer Software. Dadurch wird es, wie in [GKRS06] beschrieben, möglich, Teile einer Software, die sich gut durch ein UML-Modell erfassen lassen, direkt in einem Modell zu implementieren. Andere Aspekte werden dann durch handgeschriebenen Quellcode

ergänzt, der aber nicht in dem generierten Code integriert wird, sondern in die Generierungsquelle. Zur Bewertung der Nützlichkeit unseres Ansatzes planen wir derzeit Fallstudien, in denen das Framework zur Lösung konkreter Anwendungsprobleme genutzt wird.

Die Autoren möchten den folgenden Personen für die Mithilfe bei der Entwicklung und Implementierung des MontiCore-Frameworks, sowie die zahlreichen Verbesserungsvorschläge und Ideen danken: Christian Berger, Felix Funke, Christoph Herrmann, Tobias Höper, Christian Jordan, Ingo Maier, Patrick Neumann, Holger Rendel, Henning Sahlbach, Jens Christian Theeß, Christoph Torens und Gerald Winter.

A. Beispiele im MontiCore-Plugin

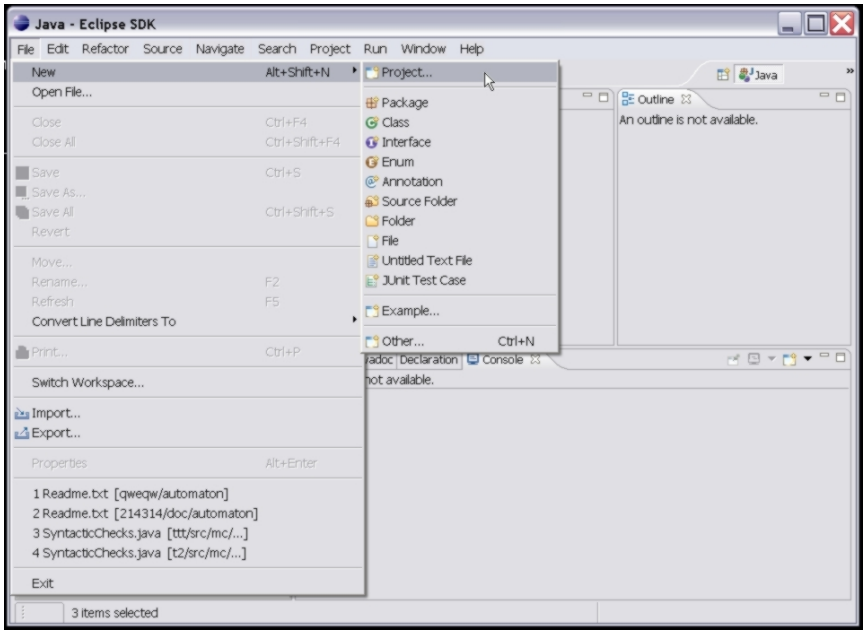


Abbildung A.1.: Anlegen eines neuen Projekts

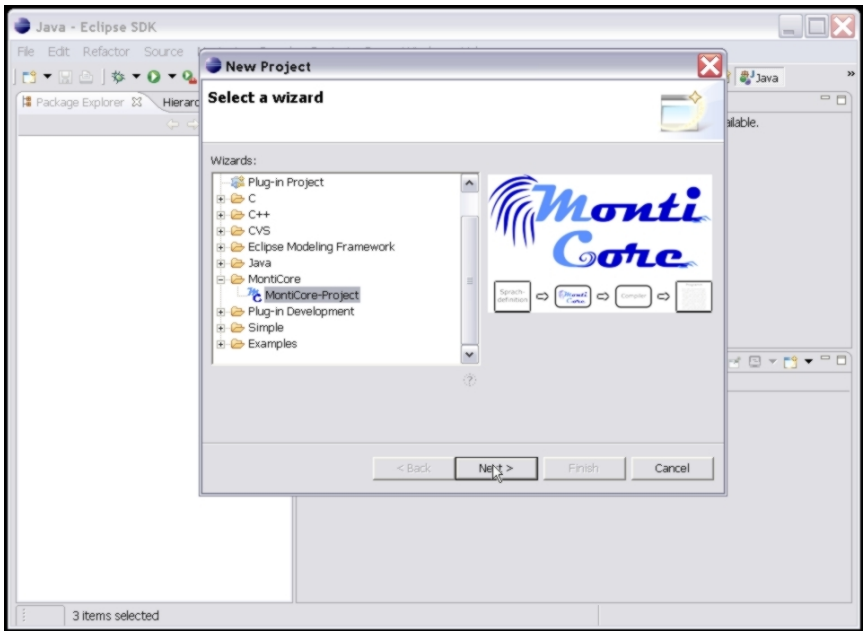


Abbildung A.2.: Auswahl eines MontiCore-Projekts

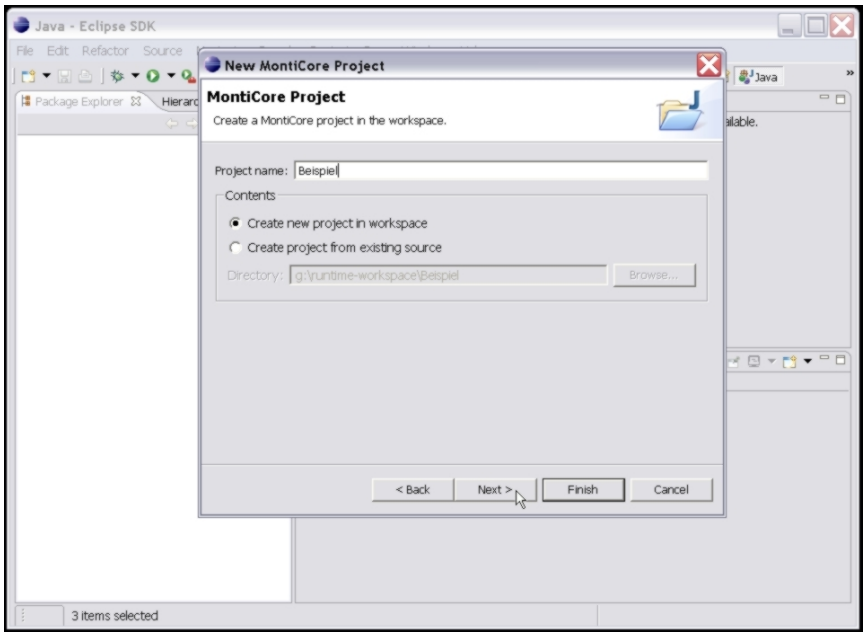


Abbildung A.3.: Namen für das Projekt vergeben

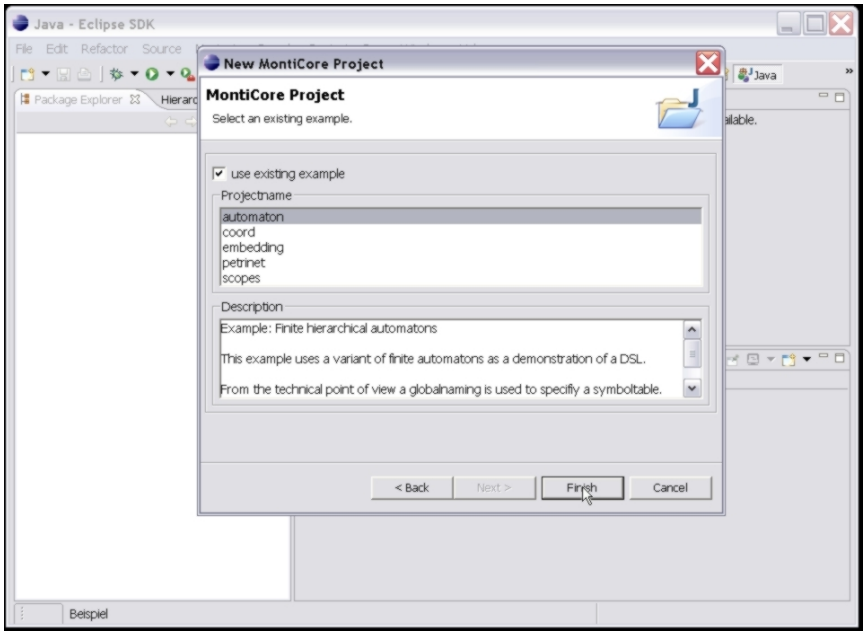


Abbildung A.4.: Auswahl des gewünschten Beispiels

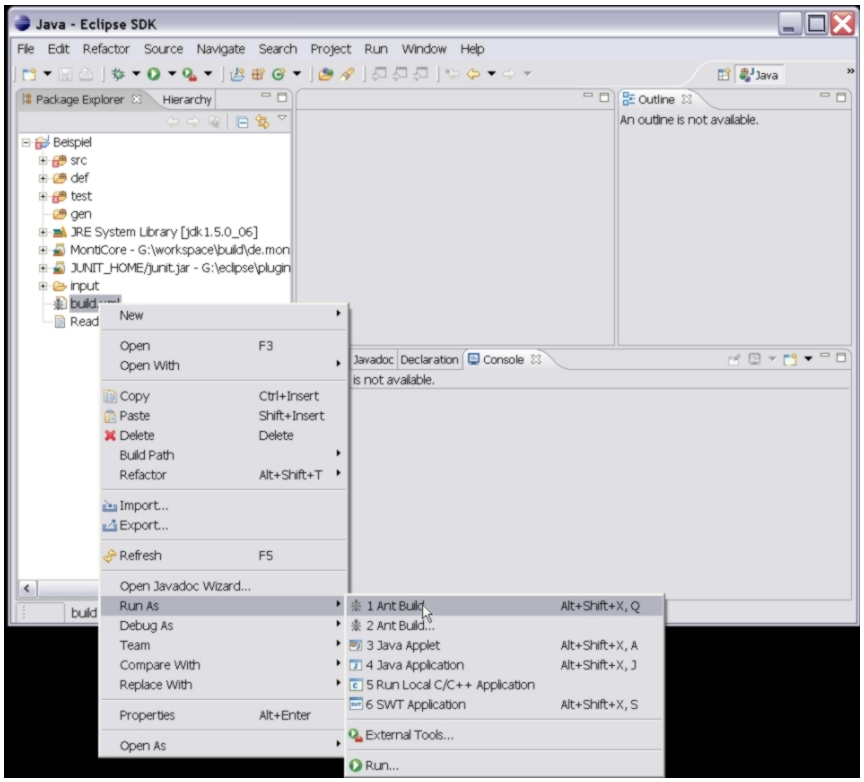


Abbildung A.5.: Auführen des Ant-Skripts

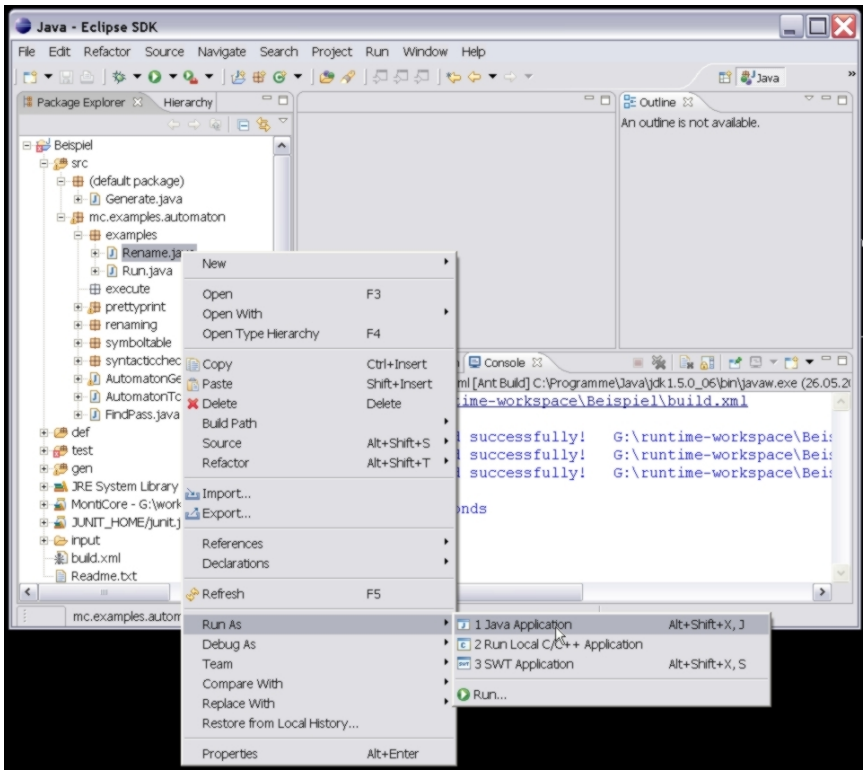


Abbildung A.6.: Ausführen des Beispiels

B. ENBF des MontiCore-Grammatikformats

EBNF

```
1  Grammar ::=
2    ('package' (IDENT | 'grammar' ) ('.' (IDENT | 'grammar' ) )* )? ';'
3    'grammar' IDENT ('extends' IDENT ('.' IDENT )* )?
4    '{' (GrammarOption | LexRule | Rule | Concept )* '}' ;
5
6  GrammarOption ::=
7    'options' '{'
8      (ParserOption | LexerOption | HeaderOption | VariousOptions )*
9    '}' ;
10
11 ParserOption ::=
12   'parser' STRING ;
13
14 LexerOption ::=
15   'lexer' STRING ;
16
17 HeaderOption ::=
18   'header' STRING ;
19
20 VariousOptions ::=
21   ('nows'| 'noslcomments'| 'nomlcomments'| 'noanything'|
22    'noident'| 'dotident'| 'nostrings')+ ;
23
24 LexRule ::=
25   'ident' ('/')? IDENT (STRING )? STRING ';' ;
26
27 Rule ::=
28   IDENT ('(' Interface (',' Interface )* ')' )?
29   ('[' NonTerminal (',' NonTerminal )* ']' )? '='
30   Alt ('|' Alt )* ';' ;
31
32
```

Abbildung B.1.: MontiCore-Grammatik im EBNF-Format

EBNF

```
1  Interface ::=
2    (Block )? IDENT ;
3
4  Alt ::=
5    (RuleComponent )* ;
6
7  Block ::=
8    '(' ('options' '{' Action '}' ':' )? Alt ('|' Alt )* ')' (('?'|'*'|'+'|'=>'))? ;
9
10 Terminal ::=
11   (IDENT '=' | IDENT ':' )? ('!')? STRING ;
12
13 Constant ::=
14   (IDENT ':' )? ('!')? STRING ;
15
16 ConstantGroup ::=
17   (IDENT '=' | IDENT ':' )? '[' Constant ('|' Constant )* ']' ;
18
19 NonTerminal ::=
20   (IDENT '=' | IDENT ':' )?
21   (('$'|'_'))? IDENT ('(' 'parameter' IDENT ')' )?
22   ('->' ('[' IDENT (',' IDENT )* ']' ) )? (('?'|'*'|'+'))? ;
23
24 Eof ::=
25   'EOF' ;
26
27 Sematicprecicateoraction ::=
28   '{' Action '}' ('?')? ;
29
30 Concept ::=
31   'concept' IDENT Concept ;
32
33 RuleComponent ::=
34 block | terminal | constantgroup | nonterminal | eof | sematicprecicateoraction;
35
36 IBlock ::=
37 block;
38
39 ITerminal ::=
40 terminal | constant;
```

Abbildung B.2.: MontiCore-Grammatik im EBNF-Format (Fortsetzung)

C. MontiCore-Grammatik im MontiCore-Grammatikformat

MontiCore-Grammatik

```
1  package mc.grammar;
2
3  /** The grammar describes the GrammarDSL in its own format */
4  grammar Grammar {
5
6  options {
7          parser lookahead=3
8          lexer  lookahead=2 "testLiterals=false;"
9  }
10
11 // Numbers
12 ident NUMBER "( '1'..'9' ) ('0'..'9')*";
13
14 /** A MontiCore grammar describes a context free grammar in an own format
15   @attribute PackageName The name of the package containing this grammar
16   @attribute Name The name of this grammar
17   @attribute Superclass Simple or qualified name of
18      the supergrammar or null if no supergrammar exists
19   @attribute Options List of options for this grammar
20   @attribute LexRules List of identifiers
21   @attribute Rules List of rules (aka productions) of this grammar
22   @attribute Concepts List of additional concepts
23 */
24 Grammar =
25   (!"package"
26     (PackageName:IDENT | !"grammar"
27       {a.getPackageName().add(new mc.ast.ASTString("grammar"));} )
28     ("."  (PackageName:IDENT|  !"grammar"
29       {a.getPackageName().add(new mc.ast.ASTString("grammar"));} ))* ";" )?
30   !"grammar" Name:IDENT
31     (!"extends" Superclass:IDENT ("." Superclass:IDENT)* )?
32   "{"
33     ( Options:GrammarOption |  LexRules:LexRule | Rules:Rule | Concepts:Concept )*
34   "}" EOF;
```

Abbildung C.1.: MontiCore-Grammatik im MontiCore-Grammatikformat

MontiCore-Grammatik

```
/** Options for a grammar
  @attribute ParserOptions List of options for the parser generation
  @attribute LexerOptions List of options for the lexer generation
  @attribute HeaderOptions List of options for the header of
    both lexer and parser
  @attribute VariousOptions List of various options for a grammar
        */
GrammarOption=
  !"options" "{"
  ( ParserOptions:ParserOption |  LexerOptions:LexerOption
  | HeaderOptions:HeaderOption | VariousOptions:VariousOptions )*
  "}" ;

/** Options for the parser generation
  @attribute Lookahead Loookahead for the parser generation
  @attribute ParserOptions Options for the parser generation
*/
ParserOption=
  !"parser" (!"lookahead" "=" Lookahead:NUMBER)? (ParserOptions:STRING)?;

/** Options for the lexer generation
  @attribute Lookahead Loookahead for the lexer generation
  @attribute LexerOptions Options for the lexer generation
*/
LexerOption=
  !"lexer" (!"lookahead" "=" Lookahead:NUMBER)? (LexerOptions:STRING)?;

/** Options for the parser generation
  @attribute HeaderOptions Options for the header of both lexer and parser
*/
HeaderOption=
  !"header" HeaderOptions:STRING;

/** Various options for the grammar generation
  @attribute Nows If set to true, no rule for whitspace/tabs etc. is created
  @attribute Noslcomments If set to true, no rule for singleline
    comments is created
  @attribute Nomlcomments If set to true, no rule for multiline comments
    is created
  @attribute Noanything If set to true, no rule for not matched
          characters is created
  @attribute Noident If set to true, no standard identifier IDENT is created
  @attribute Dotident If set to true, a IDENT allowing dots is created
  @attribute Nostring If set to true, no standard identifier
    STRING is created
  @attribute Nocharvocabulary        If set to true, no standard vocabulary
    is created
*/
```

Abbildung C.2.: MontiCore-Grammatik im MontiCore-Grammatikformat (Fortsetzung)

MontiCore-Grammatik

```
1  VariousOptions=
2    (options {greedy=true;} :
3       Nows:[!"nows"]
4       | Noslcomments:[!"noslcomments"]
5       | Nomlcomments:[!"nomlcomments"]
6       | Noanything:[!"noanything"]
7       | Noident:[!"noident"]
8       | Dotident:[!"dotident"]
9       | Nostring:[!"nostring"]
10      | Nocharvocabulary:[!"nocharvocabulary"] )+ ;
11
12 /** A LexRule is formed by the keyword ident in the beginning
13   followed by an   option slash indicating that this LexRule
14   is protected. The following name, options and symbol are handed
15   to antlr directly
16   @attribute Protected If true, this identifer is protected and
17   call only be called by other identifiers
18         @attribute Name Name of this identifier, only uppercase letters
19         should be used
20   @attribute Option Options for antlr
21   @attribute Symbol Sybol definition for antlr
22 */
23 LexRule =
24   !"ident" (Protected:[Protected:"/"])? Name:IDENT (Option:STRING)?
25   Symbol:STRING ";";
26
27 /* A rule represents a rule (production) in a context free grammar.
28   @attribute LeftSide Name of the rule
29   @attribute Interfaces List of interface for this rule
30   @attribute RuleParameters List of formal Parameters handed to this rule
31   @attribute Alts List of alternatives representing the body of the rule
32 */
33 Rule =
34         LeftSide:IDENT
35   ( "(" Interfaces:Interface ("," Interfaces:Interface)* ")" )?
36   ("[" RuleParameters:NonTerminal ("," RuleParameters:NonTerminal)* "]")?
37   "=" Alts:Alt ("|" Alts:Alt)* ";";
38
39 /* An interface ends up as an interface the associated AST-class
40   then implements.
41   The precicate block are the syntactic predicates know from antlr.
42   That only predicates are used and not normal blocks is checked by
43   syntatic check.
44   @attribute Predicate A (syntatic) predicate used in the interface rules
45         @attribute Name Name of the interface
46 */
47   Interface =
48   (Predicate:Block)? Name:IDENT;
```

Abbildung C.3.: MontiCore-Grammatik im MontiCore-Grammatikformat (Fortsetzung)

MontiCore-Grammatik

```
/* An alternative represents an alternative in a rule or block and
  contains (Rule)Components
  @attribute Components List of the rule components of this alternative
*/
Alt =
  (Components:RuleComponent)* ;

/* A block is something used in rules which is surrounded by ()
  @attribute Option options for this blook like a non-greedy behaivior
  @attribute Alts List of alternatives
  @attribute Iteration Iteration of this block
*/
Block (RuleComponent)=
  "(" (!"options" "{" Option:_Action "}" ":")?
  Alts:Alt ("|" Alts:Alt)* ")"
  (Iteration:["?"|"*"|"+"|Predicate:"=>"])?;

/* Reference to another rule, external rules start with $ which is not reflected
in the grammar, but the code generation relies on that. The parameter allows
to choose between several parsers
  @attribute VariableName Name for a variable binding
  (Note: Only one attribute of VariableName and UsuageName may have a value)
  @attribute UsageName Name for a
  @attribute Embedded
  @attribute Parameter
  @attribute RuleParameters
  @attribute Iteration
*/
NonTerminal (( (IDENT ("="|":"))? ("$"|"_")? IDENT)=> RuleComponent)=
  (VariableName:IDENT "=" | UsageName:IDENT  ":")?
  (Embedded:[Version1:"$"|Version2:"_"])?
  Name:IDENT
  ("(" !"parameter" Parameter:IDENT ")")?
  ("->" ("[" RuleParameters:IDENT ("," RuleParameters:IDENT)* "]"))?
  (Iteration:["?"|"*"|"+"])?;

/* A lexsymbol is usually something like "," */
Terminal (( (IDENT ("="|":"))? ("!")? STRING)=>
            RuleComponent,(IDENT ("="|":"))=> ITerminal)=
  (VariableName:IDENT "=" | UsageName:IDENT  ":")?
  (Keyword:["!"])? Name:STRING
  (Iteration:["?"|"*"|"+"])?;

```

Abbildung C.4.: MontiCore-Grammatik im MontiCore-Grammatikformat (Fortsetzung)

MontiCore-Grammatik

```
1  /* Strings constants have the same syntax like keywords,
2  but are something completly different. They are used within [] only */
3    Constant (ITerminal)=
4      (HumanName:IDENT ":")? (Keyword:["!"])? Name:STRING;
5
6    concept classgen ITerminal {
7      Name:IDENT;
8      Keyword:[];
9    }
10
11 /* constants are sth like keywords which one needs to know that they
12   where there, e.g. public.
13   In the ast-class they are reflected as int oder boolean isMethods
14 */
15 ConstantGroup(( (IDENT ("="|":"))? "[" )=> RuleComponent)=
16   (VariableName:IDENT "=" | UsageName:IDENT  ":")?
17   "[" Constants:Constant ("|" Constants:Constant )* "]" ;
18
19
20 /* End of file as EOF keyword */
21 Eof(RuleComponent)=
22   !"EOF";
23
24 /* Handed on as antlr action or predicate, realised by external JavaLazyParser */
25 Sematicprecicateoraction(RuleComponent)=
26   "{" Text:_Action "}" (Predicate:["?"])?;
27
28 /* The grammar can be extended by using concepts */
29 Concept=
30   !"concept" Name:IDENT Concept:_Concept(parameter Name);
31
32
33 concept classgen Rule {
34   method public String toString() { return leftSide; } ;
35 }
36 }
```

Abbildung C.5.: MontiCore-Grammatik im MontiCore-Grammatikformat (Fortsetzung)

Literaturverzeichnis

[ABRW05] A. Angermann, M. Beuschel, M. Rau, and U. Wohlfarth. *Matlab - Simulink - Stateflow, Grundlagen, Toolboxen, Beispiele*. Oldenbourg Verlag, München, 2005.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[BD04] David C. Black and Jack Donovan. *SystemC: From the Ground Up*. Springer, 2004.

[BEK97] David Byers, Magnus Engstrom, and Mariam Kamkar. The design of a test case definition language. In *Automated and Algorithmic Debugging*, pages 69–78, 1997.

[BSM+04] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Groose. *Eclipse Modeling Framework*. Addison-Wesley, 2004.

[CE00] Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming : Methods, Tools and Applications*. Addison-Wesley, 2000.

[Coo00] Steve Cook. The UML family: Profiles, prefaces and packages. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 255–264. Springer, 2000.

[Cza05] K. Czarnecki. Overview of Generative Software Development. In *Unconventional Programming Paradigms (UPP) 2004*. Springer, 2005.

[DFK+04] Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *Java(TM) Developer's Guide to Eclipse*. Addison-Wesley, 2004.

[Dou04] Bruce Powel Douglass. *Real Time UML: Advances in the UML for Real-Time Systems*. Addison-Wesley, 2004.

[DRvdA+05] A. Dreiling, M. Rosemann, W.M.P. van der Aalst, W. Sadiq, and S. Khan. Modeldriven process configuration of enterprise systems. In *Proceedings of Wirtschaftsinformatik 2005*. Physica-Verlag, 2005.

[DS03] Charles Donnelly and Richard M. Stallman. *Bison Manual: Using the YACC-compatible Parser Generator*. Trade Paper, 2003.

[FPR02] Marcus Fontoura, Wolfgang Pree, and Bernhard Rumpe. *The UML Profile for Framework Architectures*. Object Technology Series. Addison-Wesley, 2002.

[GHJV96] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1996.

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von modellen in einen codebasierten softwareentwicklungsprozess. In *Proceedings of Modellierung 2006, 22.-24. März 2006, Innsbruck, Tirol, Austria*, pages 67–81, 2006.

[Gra00] J. Grabowski. On the design of the new testing language ttcn-3. *Testing of Communicating Systems.*, 13:161–176, 2000.

[Her03] Helmut Herold. *make . Das Profitool zur automatischen Generierung von Programmen*. Addison-Wesley, 2003.

[HPM+04] Pedro Rangel Henriques, Maria Joao Varanda Pereira, Marjan Mernik, Mitja Lenic, Jeff Gray, and Hui Wu. Automatic generation of language-based tools using the lisa system, 2004. submitted.

[KR05] Holger Krahn and Bernhard Rumpe. Techniques enabling generator refactoring. In *Proceedings of Summer School on Generative and Transformational Techniques in Software Engineering (Technical Report TR-CCTC/DI-36, Centro de Ciencias e Tecnologias de Computacao, Departamento de Informatica,Universidade do Minho)*, 2005.

[LMB92] John Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly, 1992.

[LMB+01] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom andJonathan Sprinkle, and Peter Volgyesi. The generic modeling environment. In *International Workshop on Intelligent Signal Processing (WISP)*. IEEE, 2001.

[MLAZ98] M. Mernik, M. Leni, E. Avdi, and V. Zumer. A reusable object-oriented approach to formal specications of programming languages, 1998.

[MLAZ02] Marjan Mernik, Mitja Lenic, Enis Avdicausevic, and Viljem Zumer. Lisa: An interactive environment for programming language development. In *CC*, pages 1–4, 2002.

[MSU04] Stephen J. Mellor, Kendall Scott, and Axel Uhl. *MDA Distilled*. Addison-Wesley, 2004.

[MZLA99] Marjan Mernik, Viljem Zumer, Mitja Lenic, and Enis Avdicausevic. Implementation of multiple attribute grammar inheritance in the tool lisa. *SIGPLAN Not.*, 34(6):68–75, 1999.

[PQ95] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software – Practice and Experience*, 25(7):789–810, 1995.

[Rum04a] Bernhard Rumpe. *Agile Modellierung mit der UML*. Springer, Berlin, 2004.

[Rum04b] Bernhard Rumpe. *Modellierung mit der UML*. Springer, Berlin, 2004.

[WWWa] Ant Web-Seite. http://ant.apache.org/.

[WWWb] http://www.antlr.org/.

[WWWc] DSL-Tools Web-Seite. http://msdn.microsoft.com/vstudio/DSLTools/.

[WWWd] Eclipse Download Web-Seite. http://www.eclipse.org/downloads/.

[WWWe] Eclipse Modeling Framework Web-Seite. http://www.eclipse.org/emf/.

[WWWf] Emfatic Web-Seite. http://www.alphaworks.ibm.com/tech/emfatic.

[WWWg] EMF OCL Plugin Web-Seite. http://www.enic.fr/people/Vanwormhoudt/siteEMFOCL/index.html.

[WWWh] Eclipse Modeling Project Web-Seite. http://www.eclipse.org/modeling/.

[WWWi] Graphical Editing Framework Web-Seite. http://www.eclipse.org/gef/.

[WWWj] Generic Modeling Environment (GME) Web-Seite. http://www.isis.vanderbilt.edu/projects/gme/.

[WWWk] Graphical Modeling Framework Web-Seite. http://www.eclipse.org/gmf/.

[WWWl] Graphical Modeling Framework Tutorial Web-Seite. http://wiki.eclipse.org/index.php/GMF_Tutorial.

[WWWm] Grammarware Web-Seite. http://www.cs.vu.nl/grammarware/.

[WWWn] Graphviz Web-Seite. http://www.graphviz.org/.

[WWWo] http://java.sun.com/j2se/javadoc/.

[WWWp] EMF Javadoc Web-Seite. http://download.eclipse.org/tools/emf/-2.2.0/javadoc/org/eclipse/emf/ecore/package-summary.html.

[WWWq] Java SDK 5.0 Download Web-Seite. http://java.sun.com/j2se/1.5.0/download.jsp.

[WWWr] Kermata Web-Seite. www.kermeta.org.

[WWWs] http://www.latex2html.org/.

[WWWt] Lisa Web-Seite. http://labraj.uni-mb.si/lisa/.

[WWWu] MontiCore Web-Seite. http://www.monticore.de/.

[WWWv] MetaCase Web-Seite. http://www.metacase.com/.

[WWWw] MPS Web-Seite. http://www.jetbrains.com/mps/.

[WWWx] Object Management Group Web-Seite. http://www.omg.org.

[WWWy] Systems Modeling Language (SysML) Web-Seite. http://www.sysml.org.

[WWWz] Unified Modelling Language Web-Seite. http://www.uml.org.